PL-TR-94-2290

SUPPORT OF ENVIRONMENTAL REQUIREMENTS FOR CLOUD ANALYSIS AND ARCHIVE (SERCAA): DATABASE MANAGEMENT FACILITY

Charles F. Ivaldi
Joseph Doherty
Courtney C. Scott
Gary B. Gustafson

Atmospheric and Environmental Research, Inc.
840 Memorial Drive
Cambridge, MA 02139

November 20, 1994

Final Report
December 28, 1990 - September 30, 1994

Approved for public release; distribution unlimited

PHILLIPS LABORATORY
Directorate of Geophysics
AIR FORCE MATERIEL COMMAND
HANSCOM AIR FORCE BASE, MA 01731-3010

"This technical report has been reviewed and is approved for publication."

ALLAN J. BUSSEY
Contract Manager

DONALD A. CHISHOLM
Chief, Satellite Analysis and Weather
Prediction Branch
Atmospheric Sciences Division

ROBERT A. McCLATCHEY, Director
Atmospheric Sciences Division

This report has been reviewed by the ESC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS).

Qualified requestors may obtain additional copies from the Defense Technical Information Center (DTIC). All others should apply to the National Technical Information Service (NTIS).

If your address has changed, or if you wish to be removed from the mailing list, or if the addressee is no longer employed by your organization, please notify PL/IM, 29 Randolph Road, Hanscom AFB, MA 01731-3010. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
20 November 1994

3. REPORT TYPE AND DATES COVERED
Final Rpt., 12/28/90-9/30/94

4. TITLE AND SUBTITLE
Support of Environmental Requirements for Cloud Analysis and Archive (SERCAA): Database Management Facility

5. FUNDING NUMBERS
PE 62101F
PR 6670
TA 17 WU CA
Contract F19628-91-C-0011

6. AUTHOR(S)
Charles F. Ivaldi, Joseph Doherty, Courtney C. Scott, Gary B. Gustafson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Atmospheric and Environmental Research, Inc.
840 Memorial Drive
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
(none)

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Phillips Laboratory
29 Randolph Road
Hanscom AFB, MA 01731-3010
Contract Monitor: Alan Bussey/GPAB

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
PL-TR-94-2290

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A satellite database management facility was developed to support implementation and real-data testing of new multisource, multispectral satellite-based cloud analysis algorithms. Cloud algorithms were produced separately under Phase I of the Support of Environmental Requirements for Cloud Analysis and Archive (SERCAA) research and development project. The objective of the database project was to develop a unified capability to store, update, retrieve and catalog satellite and supporting data plus derived algorithm products. Data processing and storage requirements dictated a centralized database management facility capable of operating over a network of processing and mass storage systems. Based on these requirements, specifications were generated for a suitable database management facility. This was followed by development and implementation of software and hardware on the Air Force Interactive Meteorological System (AIMS) used by the Atmospheric Sciences Division at the Air Force Phillips Laboratory for SERCAA algorithm testing. Key features of the database management facility include simultaneous access to the database by multiple users, support for interactive and programmed queries, easy access to satellite and ancillary data from within application programs, network access to the database from multiple-vendor computers, and support for both on-line and removable data storage media.

14. SUBJECT TERMS
data processing   cloud analysis   database management

15. NUMBER OF PAGES
84

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYM INDEX

AIMS	Air Force Interactive Meteorological System
AVHRR	Advanced Very High Resolution Radiometer
DEC	Digital Equipment Corporation
DMSP	Defense Meteorological Satellite Program
FDL	File Definition Language
GMS	Geostationary Meteorological Satellite - Japan
GOES	Geosynchronous Operational Environmental Satellite - U.S.
I/O	Input/Output
LAVC	Local Area VAX Cluster
LZW	Lempel-Ziv-Welch
METEOSAT	Meteorological Satellite - Europe
NOAA	National Oceanic and Atmospheric Administration
OLS	Operational Linescan System
RMS	Record Management System
ROI	Region of Interest
RPC	Remote Procedure Call
RTNEPH	Real Time Nephanalysis
SAT	SERCAA North America Test Bed
SCSI	Small Computer System Interface
SDB	SERCAA Database
SDT	SERCAA Desert Test Bed
SERCAA	Support of Environmental Requirements for Cloud Analysis and Archive
SET	SERCAA Equatorial Test Bed
SGI	Silicon Graphics Inc.
TACNEPH	Tactical Nephanalysis
TCP/IP	Transfer Control Protocol/Internet Protocol
TDB	TACNEPH Database
TIFF	Tagged Image File Format
TIROS	Television and Infrared Operational Satellite
VAS	VISSR Atmospheric Sounder
VISSR	Visible Infrared Spin-Scan Radiometer
VMS	Virtual Memory System
VUP	VAX Units of Processing

1.0 Introduction

This document details development of a database management facility to support implementation and testing of new satellite-based cloud-analysis algorithms under Phase I of the Support of Environmental Requirements for Cloud Analysis and Archive (SERCAA) research and development project (Gustafson et al., 1994a).* The project objective was to provide a multispectral multisource global cloud analysis capability for use in determining the radiative and hydrological effects of clouds on climate and global change and in initializing operational cloud forecast models. Data processing requirements dictated that a comprehensive database management facility would be required to handle the tremendous volume of data that cloud analysis algorithms would require for development. These requirements included:

1. processing of sensor data from multiple satellite platforms including DMSP/OLS, NOAA/AVHRR, GOES/VAS, METEOSAT/VISSR, and GMS/VISSR;
2. spatial resolution of the data would be governed by the highest resolution at which both visible and infrared data would be globally available;
3. utilization of the full range of spectral information available from satellite sensors; and
4. collection of data from three geographically defined test-bed areas collectively covering just over half the northern hemisphere (see Figure 1) from all satellite platforms for the last 10 days of every other month beginning in March 1993 and ending in September of 1995.

This document details the design, development, and implementation of a database management facility for Phase I of SERCAA.

1.1 SERCAA Processing Overview

SERCAA is a research and development project that will provide an integrated ensemble of global cloud analysis algorithms applicable to sensor data from both civilian and military satellites for use in determining the radiative and hydrological effects of clouds on climate and global change. The SERCAA model will be the next-generation prototype for the RTNEPH. SERCAA cloud products will be available to a wide community of users. The four principal accomplishments of SERCAA are: 1) incorporation of high-resolution sensor data from multiple military and civilian satellites, polar and geostationary,

* All subsequent references to SERCAA imply SERCAA Phase I.

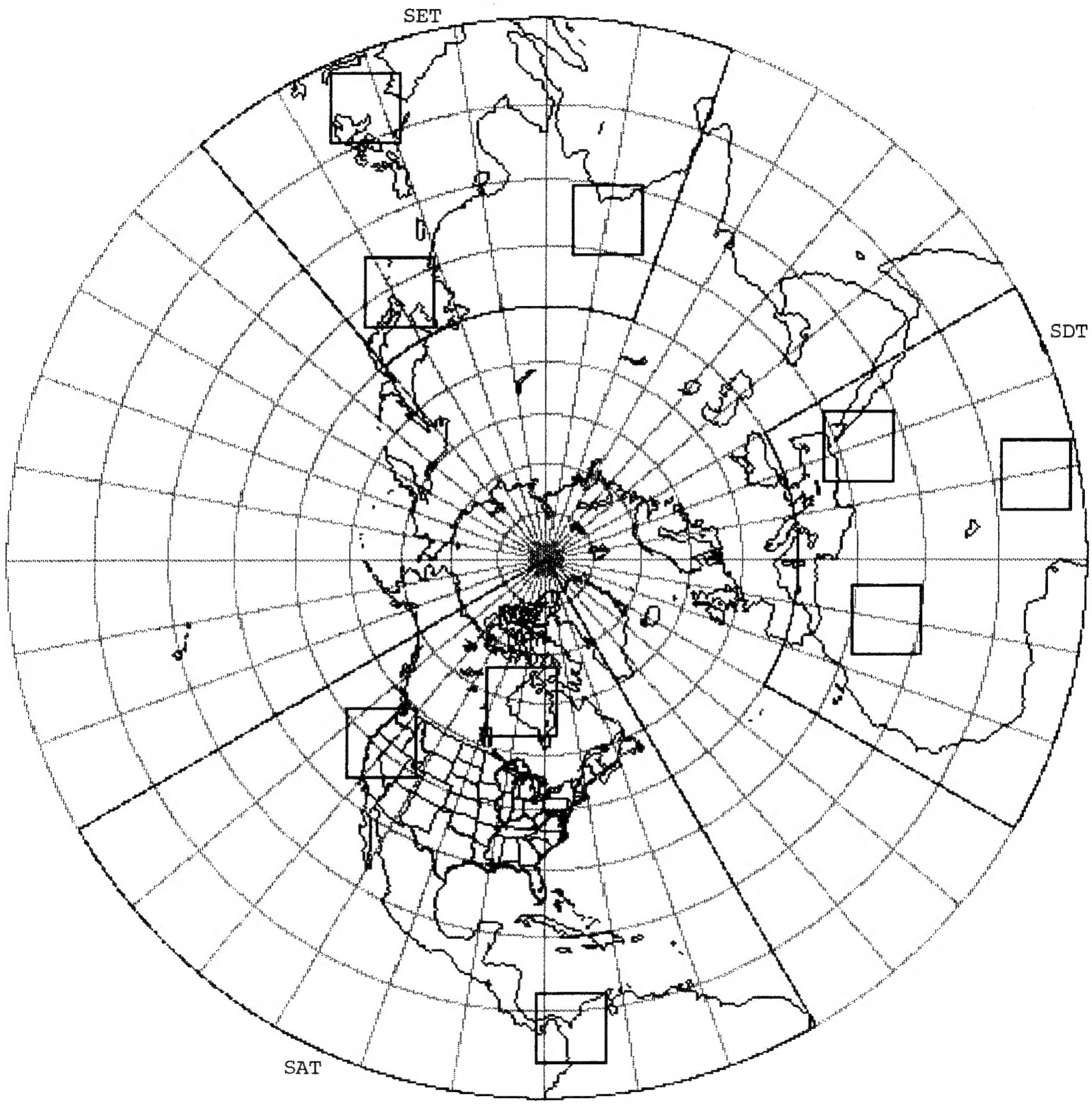

Figure 1. *SERCAA test bed areas. The smaller boxes located within the areas denote sub-areas used in the layering and integration steps.*

into a real-time cloud analysis model; 2) demonstration of multispectral cloud analysis techniques that improve the detection and specification of clouds, especially cirrus and low clouds; 3) augmented parameter, algorithm, and data base specifications for an improved cloud retrieval model; and 4) design and prototype of a global archive of these cloud analysis products in support of climate research.

The SERCAA program consists of a series of processes to analyze and integrate data from multiple satellite platforms into a single cloud analysis product. Program elements required to process raw sensor data, collected from each of the satellite platforms, into individual cloud analysis products include separate total cloud algorithms for DMSP, AVHRR, and geostationary platforms, cloud layer and type algorithms, and an analysis integration algorithm. The analyzed parameters or SERCAA data products consists of:

- cloud cover
- cloud layers
- cloud type
- cloud height
- analysis confidence level

1.1.1 AVHRR Total Cloud Algorithm

The SERCAA total cloud algorithm for the AVHRR satellite platform employs a decision tree type structure which provides the basis for a multispectral classification of scene attributes depending on such factors as scene illumination, background surface type and spectral information content. The algorithm uses multispectral signatures to identify and characterize clear and cloudy regions of the scene. Scene analysis is performed on a pixel-by-pixel basis. The AVHRR total cloud algorithm consists of a series of cloud tests that identify each pixel within an analysis scene as being cloud filled or cloud free. Each cloud test is based on a specific spectral signature that exploits radiance measurements from one or more sensor channels. The algorithm is similar to that used in the TACNEPH program.

1.1.2 DMSP Total Cloud Algorithm

The SERCAA DMSP total cloud algorithm operates on infrared and combined infrared and visible OLS data. The algorithm follows the approach outlined by Gustafson and d'Entremont (1992) for the TACNEPH program. It consists of two statistical threshold type algorithms which are designed to operate using either a single infrared thermal window channel alone or in combination with a visible channel. Both the single

channel and bispectral algorithms are designed to identify cloud filled, cloud free and partially cloudy pixels within the analysis scene. Processing of the pixels within the analysis scene is performed by first dividing it into analysis boxes whose size is dictated by application requirements. Execution of the algorithm is then performed on each analysis box comprising the scene on a pixel-by-pixel basis. The output product of the algorithm is total cloud amount for each box.

The threshold approach utilized by the algorithm allows any uncertainties in the data, including sensor calibration, clear scene characteristics, and atmospheric transmission, to be accounted for in a single threshold value. An empirically derived dynamic correction factor is used to account for all sources of error collectively without the need to understand and quantify the individual contributions. As will be explained in the sections which follow, the use of a dual threshold approach makes it possible to define partially cloudy pixels within the scene.

The performance of the algorithm is critically dependent on the ability to accurately characterize cloud free backgrounds. This is achieved through the identification of the following:

- land/water/desert boundaries
- snow and ice location
- clear scene infrared brightness temperature
- clear scene reflectance

1.1.3 GEO Total Cloud Algorithm

The SERCAA total cloud algorithm for geostationary satellite platforms employs a hybrid approach to discriminate cloud cover. Identification of cloud contaminated pixels within an analysis scene is accomplished through the use of temporal differencing, dynamic thresholding, and spectral discriminant tests. The algorithm is applicable to the following satellite systems:

- GOES (Geosynchronous Operational Environmental Satellite)
- GMS (Geostationary Meteorological Satellite - Japanese)
- METEOSAT (Meteorological Satellite - European)

The first level of processing utilizes a temporal differencing technique to identify new cloud development and existing cloud features that have moved over either previously clear background or lower cloud level. This test exploits the resultant change in infrared bright-

ness temperature and/or visible reflectance caused by these cloud features in collocated pixels taken from sequential satellite images. Cloud detection is accomplished by identifying pixels that change by an amount greater than a preset threshold over a one hour time interval. During daytime conditions, when both visible and infrared sensor data are available, a bispectral technique is employed. This technique makes a determination of cloud status by simultaneously examining the change in infrared brightness temperature and visible reflectance. A pixel is flagged as cloud filled if it is colder and brighter than the corresponding pixel from the previous collection period. Otherwise, the pixel is flagged as cloud free.

The second level of processing for pixels within the analysis scene is a dynamic threshold test. This test uses information on the thermal structure of new clouds determined from the temporal differencing test to classify surrounding pixels within an analysis area. To accomplish the dynamic threshold test, the minimum and maximum brightness temperatures of the pixels classified as cloudy by the temporal differencing test are identified. Then the maximum temperature is used to define an infrared brightness temperature cloud threshold for the remaining pixels within a defined analysis subregion. Finally, the threshold is set at the maximum temperature minus an offset to eliminate anomalous values. Currently, the offset is set to exclude the warmest five percent of the pixels within the subregion. Any pixels colder than the threshold are classified as cloudy.

The third and final level of processing exploits cloud spectral signatures. Spectral tests are only applied to pixels which have not been classified as cloudy by either the dynamic threshold test or the temporal differencing test. Spectral tests are contained within the same analysis loop as the dynamic threshold test. Thus, the spectral tests are executed on a pixel-by-pixel basis for each pixel within the subregion not yet classified as cloudy. Multiple spectral tests are available, however, the set of tests applied to any particular data set is dependent on the sensor channels available and the amount of solar illumination.

1.1.4 Layers, Types and Height

A common cloud layering algorithm is used for all geostationary and polar orbiter satellite data. The layering algorithm operates over a relatively large region (e.g., quarter- or eighth-orbit size) to maintain the integrity of cloud layers sampled over synoptic scales. Required inputs are analysis results from the DMSP, NOAA, and geostationary total cloud algorithms and the original input sensor data. Output parameters for each layer include: fractional cloud amount, type, and cloud top height. The layer algorithm is a three-step procedure. First, it employs a maximum likelihood classification algorithm to separate multispectral sensor data measurements into spectrally and/or thermally homogeneous

layers. Second, each layer is assigned a gross cloud height and a cloud type using local spatial and, if available, spectral information. The third and final step is assignment of a local cloud height at the output grid resolution.

1.1.5 Integration

The SERCAA Cloud Analysis Integration module is the final procedure in the overall SERCAA data processing flow. The algorithm integrates the separate cloud analyses produced by the DMSP, NOAA and geostationary (GOES, METEOSAT, GMS) total cloud algorithms into a single optimal database that describes the cloud conditions at a specified time. The integration technique used is a blend of rules and a simplified optimum interpolation technology (Hamill and Hoffman, 1993).

1.2 AIMS

The development and implementation of the database management facility for Phase I of SERCAA was performed on AIMS, a computer system within the Atmospheric Sciences Division of the Geophysics Directorate of the Phillips Laboratory (Ivaldi et al, 1992). AIMS consists of multi-generational Digital Equipment Corporation (DEC) VAX minicomputers and workstations, three VAX hosted imaging workstations built around Adage imaging subsystems, a GOES mode AAA ground station interfaced to an Encore 32/67 minicomputer, and NOAA/TIROS, DMSP, and METEOSAT-5 ground stations interfaced to SUN workstations. More recently two Silicon Graphics imaging workstations have been added to AIMS to initially supplement but eventually replace the aging ADAGE-based workstations. Figure 2 shows the physical configuration of AIMS. Ethernet cable physically connects all systems to provide a hardware link for computer to computer communications but the way in which systems communicate through software varies with computer type. VAX systems communicate using DEC proprietary DECnet and Local Area VAX Cluster (LAVC) networking software while SUN and SGI systems utilize the TCP/IP networking family of protocols. To allow VAX computers to communicate with SUN and SGI computers, third party software was purchased from TGV, Inc. Known as MultiNET, this software is licensed on two of the AIMS VAX computers providing them with the ability to communicate with the SUN and SGI workstations and in general, any computer that supports the TCP/IP protocol suite.

Initially, the development of a database management facility for Phase I of SERCAA was performed exclusively on AIMS VAX computers for two reasons. First, the database work was viewed as an extension to work performed earlier under TACNEPH

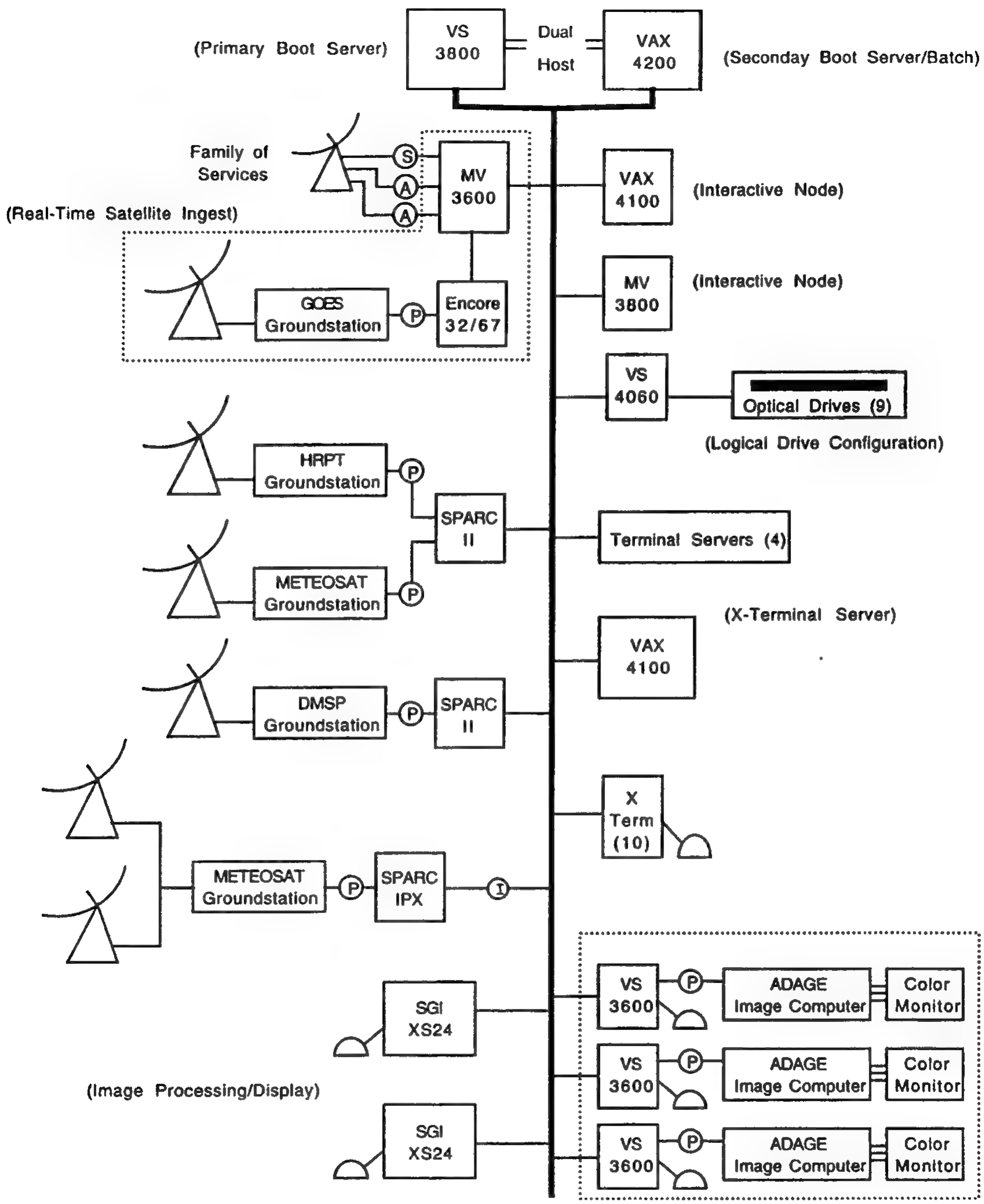

Figure 2. Schematic of AIMS hardware configuration.

(Gustafson et al., 1994b). Under TACNEPH, satellite data entered the system directly from NOAA/TIROS and DMSP direct readout ground stations as well as tape archive. Data were then moved to AIMS VAX computers. The primary benefit of working with the data on VAX minicomputers and imaging workstations was the immediate and transparent

access to TACNEPH database information and to satellite data for cloud analysis processing and visualization. This was possible because all AIMS VAX computers ran the same networking software (DEC LAVC networking software). The second reason was that at the time of TACNEPH and the beginning of work under Phase I of SERCAA AIMS was a homogeneous system, consisting entirely of VAX computers. Further into SERCAA, however, SGI imaging workstations were purchased to supplement the older and slower ADAGE-based imaging workstations. Unfortunately, the introduction of the SGI imaging workstations precluded transparent access to data for visualization because of incompatible network architectures between VAX and SGI. Part of the work performed under this contract addressed the need to reestablish transparent access to data for image visualization between AIMS VAX computers and the SGI workstations (discussed in section 7.0). The resultant work represented an important step towards extending access to the database management facility to computers outside the traditional VAX realm.

Ingest of satellite sensor data from the five satellite platforms was performed using a number of AIMS resources. Table 1 summarizes the AIMS resources used to ingest satellite sensor data from the five satellite platforms. Once ingested, data were moved to magneto-optical cartridge under VAX control and registered in the SERCAA database for subsequent cloud analysis processing. Magneto-optical cartridges were used extensively on AIMS to maintain a high volume near-online archive of satellite and ancillary data dictated by data processing requirements. Further discussion of the use of magneto-optical cartridges appears in the following section.

Table 1. Hardware summary of AIMS resources used to ingest satellite sensor data for SERCAA.

Satellite Platform	Source	AIMS Resource
NOAA/AVHRR	8mm tape	VAX 4100 with 8mm tape drive
DMSP/OLS	8mm tape	SUN Sparc II with 8mm tape drive
GMS/VISSR	4mm tape	SUN Sparc II with 4mm tape drive
GOES/VAS	Direct readout	GOES mode AAA groundstation
METEOSAT-5/VISSR	Direct readout	METEOSAT-5 groundstation

2.0 Database Requirements

Requirements for handling, storage and management of satellite data, ancillary information and cloud analysis products included the ability to maintain online or near-online multiple 10-day periods of collected raw and processed satellite data and analysis products from all satellites and locations. Table 2 shows the online storage requirements for SERCAA, stratified by satellite platform and processing level. Processing was

Table 2. SERCAA storage requirements (megabytes) for a 10-day collection period.

Processing Level/Platform	GOES[1]	Meteosat[2]	GMS[3]	DMSP[4]	NOAA[5]	Totals
Raw Data[6]	2110	1042	1042	2138	2170	8502
Intermediate Products	535	425	425	876	400	2661
Integration Products						1385
DB Entries	1680	1200	1200	600	1440	6120

[1] SAT area coverage, 4 channels @24 hours/day
[2] SDT area coverage, 2 channels @24 hours/day
[3] SET area coverage, 2 channels @24 hours/day
[4] SAT, SDT and SET area coverage, 2 channels @4 times/day
[5] SAT, SDT and SET area coverage, 5 channels @4 times/day
[6] Includes ancillary satellite information

categorized into three levels: ingest and registration of raw data, generation of total cloud and layered cloud analysis products, and generation of integrated cloud analysis products. The collected raw satellite data and associated ancillary information, requiring just under 9GB of storage, were stored on 15 magneto-optical cartridges using magneto-optical drives on AIMS (see Figure 1). The reasons for placing the raw data on optical cartridge were twofold. First, optical cartridge is removable media so that data was easily reintroduced for processing at the initial processing phase in which the data is input to producing intermediate analyses and then again at the verification phase, when the data was referenced during comparisons of the integration results with RTNEPH model output. Second, with multiple optical drive units available on AIMS, storing the raw data on optical cartridge allowed multiple simultaneous processing of the data for a different satellite, location and/or time period as well as the potential to process data from a different 10-day period.

During the requirements phase, the intermediate products were identified as the most volatile data set because they generally were no longer needed once the integration phase was completed. Although short-lived the intermediate products required high availability to satisfy the demand of processing at the integration level. For these reasons the intermediate products, requiring 2.6GB of online storage (see Table 2), resided on direct online storage provided by two 1.6GB SCSI disk drives attached to a VAX 4100. Finally, the integration step produced the smallest volume of data per 10-day period, requiring just one complete optical cartridge. The overall flow and placement of data from ingest to integration is depicted in Figure 3.

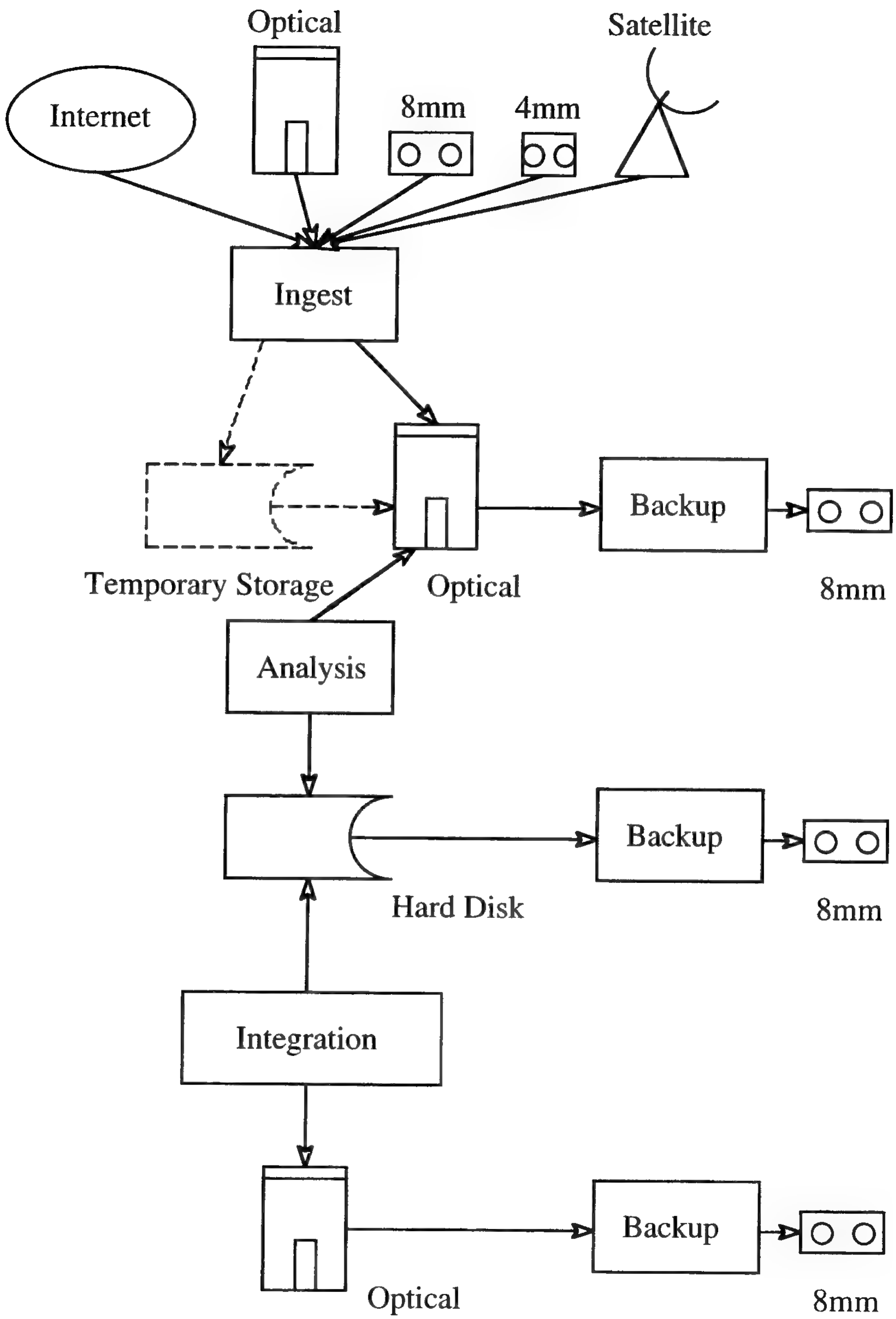

Figure 3. Data flow diagram for SERCAA.

The following were identified as functional requirements for the SERCAA database management facility:

1. Ingest of satellite data (scan line format) from multiple platforms into the database.
2. Management of data from multiple 10-day data collection periods.
3. The ability to add, delete, modify, and read database entries.
4. Interactive and programmed access to the database.
5. Interactive and programmed access to data files.

3.0 Database Specifications

The functional requirements for the SERCAA database management facility closely paralleled the TACNEPH database (TDB) effort (Gustafson et al., 1994b) but on a larger scale. Because the projected number of entries for a 10-day period was comparable with current entries managed by the TDB, a review of TDB was performed to determine the feasibility of its use for SERCAA. At the core of the TDB was a set of interrelated data dictionaries containing meta data with an organization based on the relational data model and implemented as indexed files.

The ability to track and manage the volume of data represented by a 10-day period is largely dependent on the ability to efficiently manage meta data, that is, the information that describes the data and allows users and applications to query and manipulate the data in meaningful ways. The last row in Table 2 lists the number of unique pieces of information (entries), stratified by satellite, that any database is required to maintain from a 10-day data save.

Analysis of the organization of the TDB was concluded with the recommendations that the existing structure of the TDB data dictionaries, with minor alterations to improve query optimization, would be adequate for SERCAA. Further analysis of the TDB revealed that outstanding performance problems were linked to file I/O bottlenecks tied directly to program design and the lack of file maintenance. The problems discovered during the analysis of the TDB and implemented solutions are outlined below:

1. Fragmentation of the data dictionaries.

The files were fragmented physically on disk and internally within the index structure of the file. The fragmentation was a result of poor file design. The solution was to specify file attributes that more closely reflected file use than could be obtained using default attributes when files are created. This could be accomplished using VMS File Definition Language (FDL) files to define tailored attributes set by a user. Also periodic tuning of the files using VMS-supplied services should be performed so that file design parallels file use.

2. Applications sometimes had to wait to access data dictionaries even if only to read them.

This was because default file sharing options were used to open files, which under VMS means exclusive read/write access. Others must wait until the first application that opened the file closes it. The solution was to have all applications open data dictionary files for read/write sharing. This would allow multiple applications to open and read from or write to the file simultaneously. VMS RMS services will automatically arbitrate locking at the record level to preserve file integrity. Operations in which records are read without the intent to modify or delete can explicitly unlock the record to quicken its availability to competing programs.

3. Database queries were sometimes slow.

Unless the query is based on the entry number, records must be read singly and individual fields of the record used to match query criteria. The solution was to have the entire data dictionary read into local memory for read-only operations such as a database query. Queries can than proceed at memory access speeds rather than at disk speeds. Tests in debug mode showed that a 6000 entry data dictionary composed of 100-byte records could be read into memory in about 4 seconds wall clock time.

4. Gathering entry numbers to make inserts was slow at times.

This was because unused entry numbers are found by attempting to read a record with an entry number as the primary key. If the record is not found then the key is available for use. Although this technique works it is extremely time consuming. The solution was to explicitly manage the use of entry numbers using a bit mask (refer to section 4.2) and use VMS run-time library bit-manipulation routines to provide efficient management of the bit mask.

SERCAA Phase I requirements specified the collection of data from all satellite platforms for the last 10 days of every other month beginning in March 1993 and ending in September of 1995. With the potential for having to manage in excess of 100,000 database entries, there was concern that a single set of data dictionaries would be overtaxed based on previous experience using the TDB that managed on the order of 10,000 entries. To insure that database entries could be kept to a manageable level, specifications called for a separate set of data dictionaries for each 10-day period (about 7000 entries). To facilitate movement among sets of data dictionaries (representing different 10-day periods) references to database files would employ a feature of the VMS operating system that allows one to map a physical file name to a logical name and to dynamically change the mapping at will.

4.0 Database Implementation

The SERCAA database is composed of four distinct yet interrelated components. The components are 1) a set of meta files that describe SERCAA data, 2) a companion meta file that is sorted on information that is useful for performing database queries, 3) a bit mask used to assign a unique entry number to every meta file entry, and 4) the actual satellite data, derived products and ancillary information.

4.1 Data Dictionaries

The data dictionaries contain information that describe the salient characteristics of SERCAA data. Figure 4 is a representation of the set of data dictionaries used to describe SERCAA data. At the top of the hierarchy is a core data dictionary, so called because it contains a minimum or core amount of meta data sufficient to describe the data often without the need to access other dictionaries. The core data dictionary contains a set of abbreviated entries (records) whose expanded contents are located in lower level dictionaries. These lower level dictionaries contain detailed information that describe satellite image data, derived products, and supporting data including latitude-longitude data, satellite-solar geometry information, and satellite ephemeris data. Every core data dictionary entry can be uniquely identified by an entry number. Every entry number is a member of the data dictionary entry that it identifies and it also serves to organize the data dictionaries on disk files. This organization is accomplished by implementing the data dictionaries as indexed files and using the entry number as the primary key. On VAX systems, the index file organization is directly supported by the VMS operating system.

As illustrated in Figure 4, data dictionaries are organized based on the relational data model. There is a one-to-one relationship between entries in the core data dictionary and the expanded version in each of the remaining data dictionaries (the product data dictionary actually shares a many-to-one relationship with the core data dictionary, this is addressed later in the text). For example in Figure 4, core entry number 1, an entry of type "image" shares a one-to-one relationship with entry number 1 in the image data dictionary. Thus one only need to query the core data dictionary to find references to all other data dictionaries entries that describe a particular data entity. Entries from each of the subordinate data dictionaries are often referred to collectively in the context of working with the data as a group of entries or an entry group. Tables 3 through 7 list the structure contents of each of the data dictionaries.

Data for a particular date-time, location, and satellite is represented by a group of entries whose total varies according to the number of sensor channels the satellite platform supports. The group will always contain one entry for a latitude-longitude file, one entry

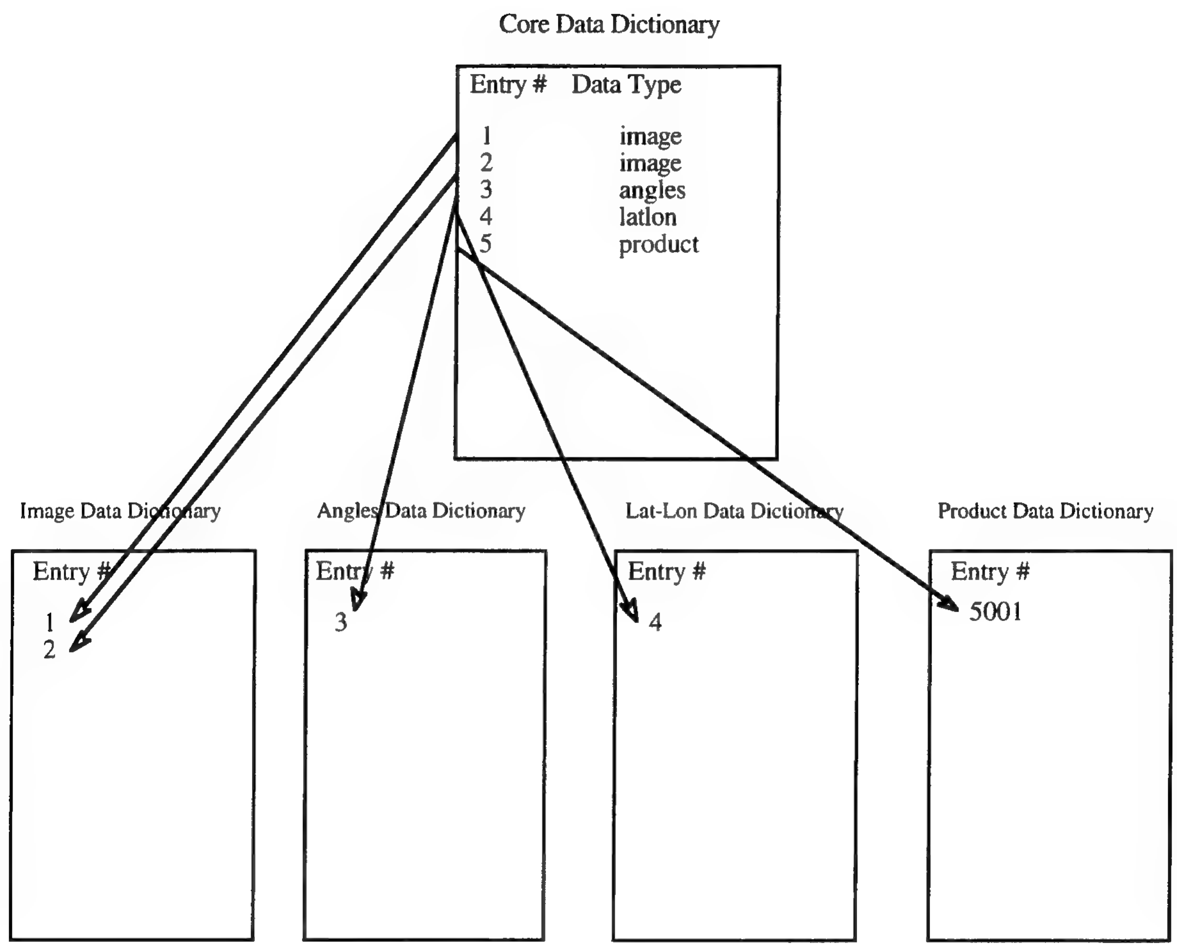

Figure 4. Representation of the relational-hierarchical set of data dictionaries that comprise the SERCAA database.

Table 3. Structure of core data dictionary entry.

Data Type	Name	Description
I*2	Entry	Data dictionary entry number
I*2	Data_Type	Data Type: 0 - unknown 1 - grayshade image 2 - ephemeris 3 - latitude-longitude 4 - angles 5 - calibration 6 - subimage 7 - product
I*2	Sat_Code	Satellite code
I*2	Sensor_Code	Sensor code
I*4	DD_Mask	Mask for related dictionary entries
I*4	Backup_Code	Backup status code
I*4 (2)	Time_Stamp	Time entry was created or modified
I*4	Version	Version number
I*4 (2)	Zulu_Time	Nominal year, julian day, & time of data
C*3	Country_Code	3-letter id region of interest
C*6	Ch_case	Case day in CYYDDD format
C*60	Data_File	Full pathname of data file

Table 4. Structure of image data dictionary entry.

Data Type	Name	Description
I*2	Entry	Data dictionary entry number
I*2	Sat_Code	Satellite code
I*2	Sensor_Code	Sensor code
I*2	Image_Type	Image calibration type
I*2	Ephem_Entry	Related entry number of ephemeris data
I*2	Latlon_Entry	Related entry number of lat-lon data
I*2	Angles_Entry	Related entry number of angles data
I*2	Calib_Entry	Related entry number of calibration coefficient data
I*2	Row_Offset	Row offset of subimage from main image
I*2	Col_Offset	Column offset of subimage
I*2	Map_Row_Offset	Row offset of lat-lon data to image
I*2	Map_Col_Offset	Column offset of lat-lon data to image
I*2	Elem_Per_Line	Number of data elements per line
I*2	Bytes_Per_Element	Number of bytes per data element
I*2	Num_Lines	Number of image lines
I*2 (10)	Related_Ch_Ent	Entry numbers of related channel entries
I*4 (2)	Time_Stamp	Time entry was created or modified
I*4 (2)	Zulu_Time	Nominal year, julian day & time of image
I*1	Num_Rel_Ch	Number of related channels
R*4	Spa_Res	Spatial resolution

Table 5. Structure of latitude-longitude data dictionary entry.

Data Type	Name	Description
I*2	Entry	Data dictionary entry number
I*2	Latlon_Code	Satellite/Sensor code for lat-lon data
I*2	Record_Length	Record length of lat-lon file
I*2	Number_Lines	Number of lines
I*2	Elem_Interval	Element interval
I*2	Line_Interval	Line interval
I*2	Samples	Number of samples per line
I*4 (2)	Time_Stamp	Time entry was created or modified
I*4	Shift	Offset to first lat-lon fiducial

Table 6. Structure of angles data dictionary entry.

Data Type	Name	Description
I*2	Entry	Data dictionary entry number
I*2	Latlon_Code	Satellite/Sensor code for angles data
I*2	Record_Length	Record length of angles file
I*2	Number_Lines	Number of lines
I*2	Elem_Interval	Element interval
I*2	Line_Interval	Line interval
I*2	Samples	Number of samples per line
L*4	Viewing	Viewing angle boolean
L*4	Zenith	Zenith angle boolean
L*4	Azimuth	Azimuth angle boolean
I*4 (2)	Time_Stamp	Time entry was created or modified
I*4	Shift	Offset to first angles fiducial

Table 7. Structure of product data dictionary entry.

Data Type	Name	Description
I*4	Entry	Data dictionary entry number = Core Entry Number * 1000 + n
I*4 (2)	Time_Stamp	Time entry was created or modified
I*2	Analysis_Type	Type of analysis product
I*2	Prod_Rec_Len	Product file record length
C*10	Username	User name of product owner
C*80 (10)	Product_File	Pathname of product file
R*4 (10,5)	Input_Params	Analysis dependent input

for satellite-solar geometry, one entry to represent products, and a variable number of entries for image files. One exception to the group definition is GOES data. This platform substitutes an ephemeris entry for the latitude-longitude entry because transformations between GOES satellite coordinates and earth coordinates can be computed directly using coordinate transformation codes. The ephemeris entry for GOES simply references a file containing satellite orbit and attitude parameters required to perform the transformation.

As mentioned previously, the product data dictionary shares a many-to-one relationship with the core data dictionary. This is because potentially many products can be derived from an image or set of images of a particular date-time, satellite, and location. This means that within the product data dictionary multiple entries need to be "reserved" so there will be ample space to describe multiple products as they are generated. To accomplish this the base or starting entry number within the product data dictionary is computed by:

$$\text{Base Entry} = \text{Core Product Entry} * 1000 + 1$$

All succeeding entries (related to the core product entry) added to the product data dictionary are incremented by one starting at the base entry.

4.2 Bit Mask

At the time satellite data were registered in the SERCAA database, new entry numbers needed to be assigned so that the data would be uniquely identified. New entry numbers were generated using a bit mask. The bit mask is logically a stream of contiguous counted bits, one for each possible entry number up to a maximum of 16K bits. Bits with

a value of one represent allocated entry numbers while bits with a value of zero represent unallocated entry numbers. To allocate one or more entry numbers, the mask is traversed from the beginning until N (N is the desired number of entry numbers) contiguous unallocated bits are found. Entries deleted from the core data dictionary automatically clear the corresponding bits in the bit mask. Deleted entry numbers are reused if the number of entries requested is less than or equal to the number of entries previously deleted. Direct use of the bit mask is normally confined to programs that ingest and register incoming data. Delete functions access the bit mask transparently. Additionally, the bit mask can be used to assist in the reconstruction of a corrupted master meta file by providing a list of known allocated entry numbers that can be used by secondary meta files to cross-reference internal entries. In the case of a corrupted or otherwise unusable bit mask, the core data dictionary can be used to reconstruct the bit mask.

On AIMS the bit mask was implemented as a sequential file with fixed length records. Software that manipulates the bit mask utilizes VMS built-in functions that directly access hardware and machine bit manipulation instructions for increased performance. Access to the bit mask utility was through the program callable function SDB_BITMASK. Refer to Appendix A for a description of this routine.

4.3 Sorted Core Data Dictionary

The data dictionaries are all accessible as long as the user or application has prior knowledge of the entry number(s) associated with the data of interest and will depend solely on SERCAA database software to retrieve entries. The ingest process is a good example of when users do not know the entry numbers associated with data they may be interested in processing because entries are just being assigned at the time of ingest. In instances such as this, users generally want to query the database using such search parameters as the date-time, satellite, and/or location of the data and receive back entry numbers that match the criteria. To facilitate such a query, a new core data dictionary was developed. This data dictionary is composed of a subset of information extracted from the core data dictionary that relates directly to query information most often posed by users; namely date-time, satellite, and location. By triply sorting this information first on date-time, then satellite, and finally location, queries can be optimized using the binary search method for sorted lists.

4.4 Satellite Data

Satellite data, derived products and supporting information generally resided on either hard disk or magneto-optical disk as sequential files with fixed-length records that corresponded to the number of pixels in a satellite scan line. Record size was documented in the corresponding image data dictionary entry. Efforts were also focused on the need to make efficient use of disk storage due to the enormous amount of space SERCAA data files occupied. This effort resulted in a change in file format for image data from essentially a raw format to the Tagged Image File Format (TIFF) as well as a change in file organization from sequential to stream. Using a TIFF software support package from Silicon Graphics Inc. that was installed on AIMS, images were compressed and decompressed using the Lempel-Ziv-Welch (LZW) compression algorithm (Welch, 1984). After being applied to image data, compression ratios of 2 to 1 were commonly achieved.

Data file names followed a convention that somewhat described file contents and duplicated the information provided by the data dictionaries. In the possible but unlikely case of a corrupted data dictionary, use of the naming convention would facilitate identification of data files for use as well as provide information to contribute to the reconstruction of a corrupted data dictionary.

4.5 Access to Multiple Databases

As mentioned previously each 10-day save set was associated with a unique set of database files to keep the number of database entries manageable. A set was composed of 5 data dictionary files (one each for core, image, latitude-longitude, angles, and product information). To move easily among sets of database files VMS logicals were used. All references to data dictionary files are specified as logical names that the operating system subsequently translates to a physical path name (i.e., device, directory, and file name). The advantage of using logical names is that if a program is coded to access a logical name then the user can change the physical data files that the program accesses simply by changing the definition of the logical names (either interactively or within a command file). This is in contrast to time consuming code changes that would be required each time the user wanted to access a different set of files. To automate the mapping of logical to physical names a convention was adopted to name physical files based on the month and year of a 10-day save set they represent. This in turn, facilitated the development of a command file users could execute to select any 10-day save set of interest simply by specifying the month/year combination of the save set the user was interested in accessing. Within the command file, the input date was used to generate the physical file names followed by the assignment of standard logical names (recognized by all database programs

and routines) to the physical file names. The database user could execute the command file as often as needed during a session to switch between online sets of database files.

5.0 Run-Time Library

The functional requirements outlined above were satisfied through development of routines that collectively define the SERCAA run-time library. The library is composed of three major components: 1) a set of primitives that allow applications to retrieve, add, delete, and modify data dictionary entries, 2) a routine that allows applications to retrieve all or part of an image, and 3) a routine that allows applications to retrieve entries from the sorted core data dictionary based on query parameters commonly used by applications. The run-time library is a VMS shareable image library. A shareable image library allows multiply concurrent applications to share a single copy of the library, and thereby conserve memory space as opposed to each application having a separate copy cataloged at link time. Shareable image libraries also eliminate the need to re-link applications when changes are made to library modules because module entry points remain fixed through use of an intermediate vector table.

5.1 Primitives

The database primitives are software routines that manipulate the data dictionaries directly. These routines either retrieve, add, update, or delete a single entry at one time so that to perform an action on multiple entries requires multiple calls to one of the routines. All data dictionary files are opened for shared read/write access so routines first perform checks for locked records and institute timed waits in the event an I/O operation is blocked because of an existing operation that has not yet completed. All routines were written in FORTRAN and use FORTRAN I/O to perform open, read, write, and re-write operations. File open operations are performed only once on the first call to the primitive to eliminate unnecessary overhead in repetitive open/close operations. A data dictionary file is closed by indicating through a calling argument that the call to the primitive represents the final call. Functional descriptions of the primitives are provided in Appendix B.

5.2 Satellite Data Access Routine

During Phase I of SERCAA an additional requirement was generated for a programmable routine that would facilitate the use of satellite data within application programs. The general requirement was to isolate the detail of I/O involved in reading satellite data to a level below the application so that the application would simply be presented with the data for an entire scene or a subset of the scene. Specific requirements

were driven by the different ways applications would want to access one or more satellite images. These included:

1. access to the entire image at one time;
2. random access to multiple scan lines of the image; and
3. access to subsets within an image scene. Access may be geographically sequential or random.

The application would also not have to be concerned with whether or not a file was compressed; the routine would determine if a file was compressed and if so present the data uncompressed to the application. Finally, the routine would have to support statistically declared buffer space to accommodate programming languages that cannot reference memory addresses contained within programming variables directly (e.g., FORTRAN).

Based on the above requirements a routine was developed and incorporated into the run-time library. The routine, SDB_GET_DATA, supports access to an entire image, random access to multiple scan lines within an image, and access to user-defined subsets within an image scene. Internal to SDB_GET_DATA a working buffer is used to store data initially followed by memory to memory copies that fill a user supplied buffer based on the data requested. The only exception to the copy operation is when a user requests an entire image; in this case the user buffer is the working buffer. SDB_GET_DATA will manage multiple files that may be opened by users making multiple calls to the module within the same program. This is accomplished by maintaining an internal link-listed data structure containing entries for each file. Entries contain file and image characteristics, information about the data request, and the location of internal working variables and buffers. A functional description of SDB_GET_DATA can be found in Appendix C.

5.3 Database Query Routine

Another requirement related to database access was the ability to allow application developers to query the database for entries based on image date and time, satellite type, and geographic location. To facilitate the query capability, a new secondary core data was used to optimize query searches. This new dictionary was created by extracting image date and time, satellite type, and geographic location information from the primary core data dictionary and sorting the elements in increasing order. This way queries would be reduced to relatively simple searches on an already sorted list that would be maintained along with the other data dictionaries. If changes were made to the primary core data dictionary, the secondary dictionary would be updated as required.

The first step towards developing a program to generate a sorted core data dictionary was to select an appropriate sorting algorithm. The selection of a suitable sorting algorithm ideally requires knowledge of the number of entries that will be sorted as well as a measure of the spatial separation of entries in the unsorted list. For SERCAA, the number of entries varied, but in general, grew at a rate governed by data availability. The total number of entries for a 10-day save period was estimated at about 7000. The spatial distribution of entries based on date-time (the primary sort parameter) within the primary core data dictionary file is a reflection of data that are registered as it becomes available. Typically chronological order is maintained within a group of contiguous entries but is almost never maintained when spanning groups. The result is that the primary core data dictionary is subject to a relatively high spatial separation of the primary sort parameter. Based on these two findings: 1) a large number of entries and 2) high spatial separation of the primary sort parameter, the Shellsort (Shell, 1959) algorithm was chosen. This algorithm attempts to reduce high spatial resolution in the early stages of the sort so that interchanges in later stages will be minimized. The interval between compared elements of a Shellsort is gradually reduced to one at which point the sort is simply interchanging adjacent elements.

Using the Shellsort algorithm, a program was developed to sort the core data dictionary. The program reads the entire data dictionary into memory and generates an unsorted list of records containing date-time, satellite, location, and entry number. The dictionary list is then sorted on date-time. Next a group of duplicate date-times is identified and further sorted on satellite. Finally, duplicate satellite groups are identified and sorted on location. This process is then repeated until the entire list is triply sorted. Test runs of the program executing on 32VUP VAX 4000-100 showed that a 7000 entry core data dictionary could be sorted in just under one minute elapsed time.

With a sorted core data dictionary in place, development of a query routine based on the functional requirements outlined above could proceed. The resultant routine, known as SDB_QUERY_CORE, is callable from FORTRAN and 'C' and takes four arguments. The major input argument is a structure that describes the query criteria for gathering entry numbers. Criteria parameters include start and stop date-time, satellite identifier, and location identifier. Parameters may be used singly or in combination with unspecified parameters interpreted as wild card specifications. The starting date-time does not necessarily have to match an existing entry in the sorted list. By setting a match-flag field within the query structure, the calling program can specify that the closest entry or the closest entry that doesn't exceed the starting date-time be returned. The last field of the query structure allows the calling program to control the amount of information returned.

By default, the query function returns a reference to a structure containing entry numbers. However, by setting the last field of the query structure to Boolean TRUE the function will also return addresses that point to the original full core data dictionary entries that coincide with the entry numbers. This is possible because SDB_QUERY_CORE reads the entire core data dictionary into memory using an optimized I/O procedure and maintains memory references to all entries. This feature allows the calling program to immediately retrieve additional core data dictionary information without the overhead of calling formal I/O routines to read the entry structures from the database. It also means that multiple calling programs can potentially share a single copy of this memory-based core data dictionary to minimize memory usage. A description of SDB_QUERY_CORE can be found in Appendix D.

5.4 Database Utility Program

With the primitives and a database query routine fully implemented, an interactive program was developed that would provide users interactive access to data dictionary information based on either known entry numbers or query parameters. The program, known as SDB, allows the user to query data dictionaries for entries based on combinations of time or time range, satellite, and location. SDB returns information from one or more dictionaries that is integrated and formatted as a single line of output to the screen or to a file. For individual entries, an expanded output is produced listing all dictionary information for the dictionary type specified. Appendix E provides interactive help-level documentation on the SDB program.

6.0 Streamlined Ingest and Registration Procedures

Ingest procedures used early on in SERCAA were cumbersome to use, required a high level of operator interaction, demanded code changes if data were targeted for different storage areas, and could not accommodate manual override actions in cases where they were necessary (e.g., computer failure). New goals were set to streamline the ingest and database registration procedures for all satellite platforms so that data could quickly and efficiently be made accessible to cloud analysis applications in a timely manner. The goals were to 1) minimize operator interaction and maintenance through automation, 2) add flexibility to the procedures so that data could be located and identified anywhere within AIMS, and 3) facilitate recovery procedures when the automated procedure was interrupted.

As implemented, the streamlined ingest and registration tasks utilize a procedural approach common to all satellite platforms. This approach is applicable to ingest (defined

as all the processing needed to prepare the data for registration) and registration (defined as processing required to move satellite data to a storage area that will subsequently be used for all further processing (the destination) and then register the data in the database) of satellite data from all platforms and can be described as a series of steps. First, satellite data are received either directly through an AIMS satellite ground station computer or on tape from an archive center. The data are then processed by an ingest function specific to the given platform. The ingest function creates source files (an information file, a latitude-longitude data file, an angles data file and a satellite image data file for each sensor available) from the raw data and invokes the registration process. To minimize operator intervention the ingest function utilizes an input parameters file to store parameters that change periodically. Information such as the location of the raw data, the location of the source and destination files, and platform-dependent information are stored in this file (see Appendix F for a description of the contents of the input parameters file). The registration process moves the source data files to the destination area and registers the data in the database. Source and destination areas (described in the input parameters file) as well as data needed to generate meta data are documented in an information file known as the SDB information file (see Appendix G for a description of the contents of the SDB information file). Because source and destination areas for files are specified in these two files, changing storage conditions on AIMS were easily handled by simply making changes to the input parameters file. Figure 5 is a schematic of the ingest and registration process, showing the interrelationships between processes and files.

Failures during ingest and registration do occur. Failures may be due to such things as unreadable tapes, lack of available disk space, and hardware failures. The design of the common procedural approach explicitly addresses recovery actions that can be taken when failures such as these occur. In general there are three key processing points where recovery measures can be instituted. The first is when raw data are being piped into the ingest process. For example, there may be thirty passes on a data tape being ingested and registered. Say, an error occurs during the 15th pass and the process aborts. At this processing point the operator can intervene and restart the process beginning with pass number 15. The second point is during the invocation of the registration process. If a failure occurs here the registration process can be invoked manually. Finally, recovery is possible during registration. Because entry numbers assigned during registration are

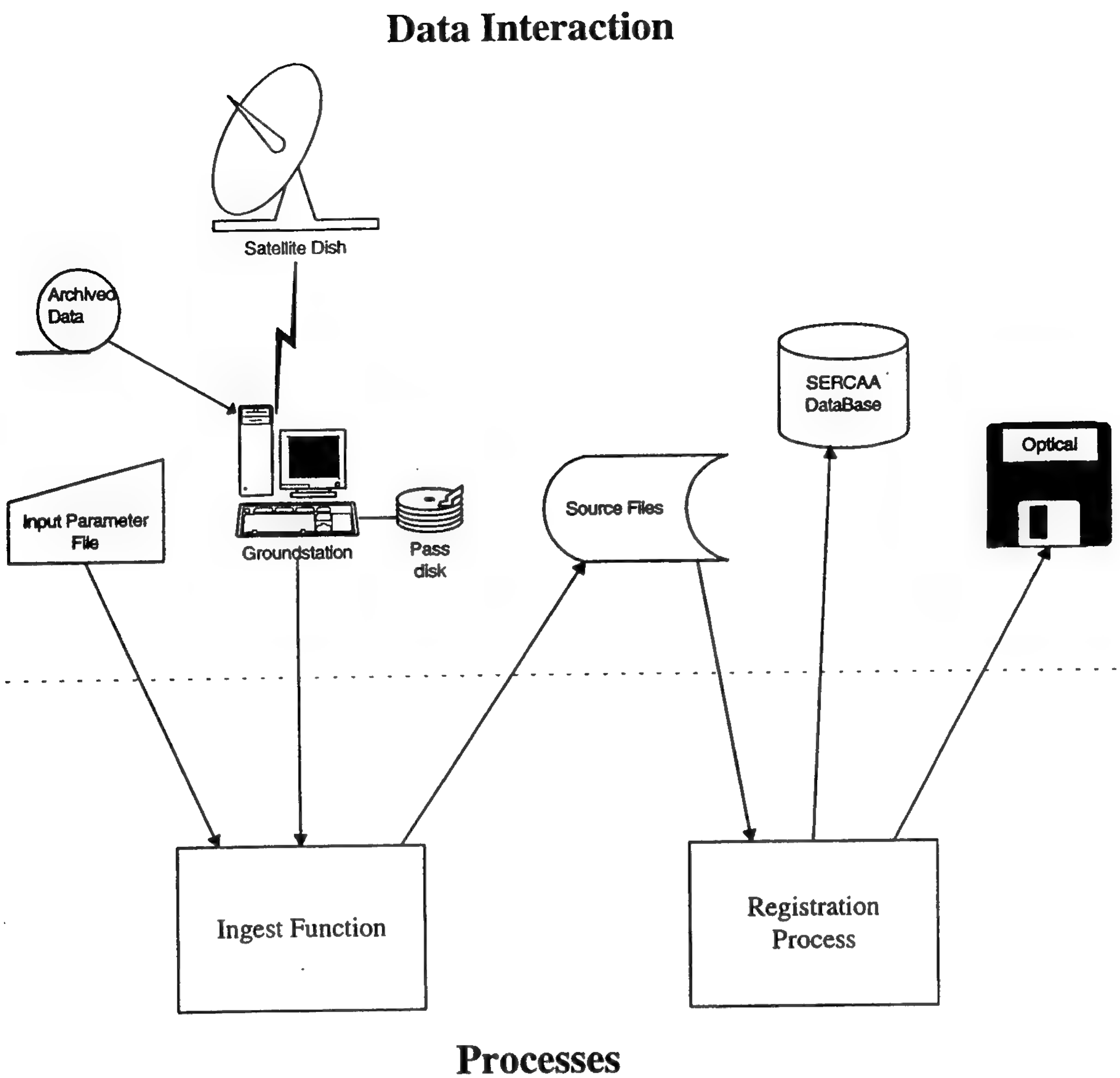

Figure 5. SERCAA ingest/registration.

documented in the SDB information file, references to aborted entries can be removed from the database and the registration can be restarted manually.

A file naming convention was adopted as part of the streamline task to allow the satellite data files generated during the ingest and registration process to be named in a unique and self-describing manner. The following summarizes the convention used:

1. All file names are fixed length, 18 characters.
2. Fields are separated by an underscore for clarity.
3. The first three characters identify the satellite.
4. The next three characters identify the type of data.

Numerics (001 - 00N) identify satellite sensor data.

LAT_ identifies latitude-longitude data

ANG_ identifies angles data

SDB_ identifies the SDB information file

5. The next 3 characters identify the geographic region of interest.
6. The next 3 characters identify the Julian day.
7. The final 2 characters identify the hour of the data.
8. All file extensions are 3 characters in length separated by the period character.
9. The extension for satellite image data will be either DAT for raw format or TIF for TIFF format.
10. The extension for all other data is DAT.

Example:

F11_001_SET_145_00.TIF

identifies channel 1 visible data from the DMSP F11 satellite for Julian day 145 at 0Z covering the predefined equatorial ROI for SERCAA. The extension indicates the file is in TIFF format.

In addition to adherence to the file naming convention, disk and directory structures are defined to provide consistency in file location. Generally, all satellite imagery and supporting data files are stored on optical disk. All opticals are initialized before use and are given a label of six characters. The first three characters identify satellite/sensor platform while the remaining three are used for a sequence number. Figure 6 illustrates the directory structure placed on opticals and hard disk storage. During registration sub-directories of the directory named "DATA" (see Figure 6) are created and then populated with the appropriate data files. The name of the directory is derived from the date of the data found in the SDB information file. The format used to name the subdirectory is in the form CYYJJJ, where C is century, YY is the year, and JJJ is the Julian date. Used in combination, the file naming convention and directory structure ensure each data file is uniquely identified.

The SDB information file and the self-describing file naming convention provide the SERCAA database administrator with the information necessary to reconstruct part or all of a database if it becomes corrupted. The SDB information file contains needed meta

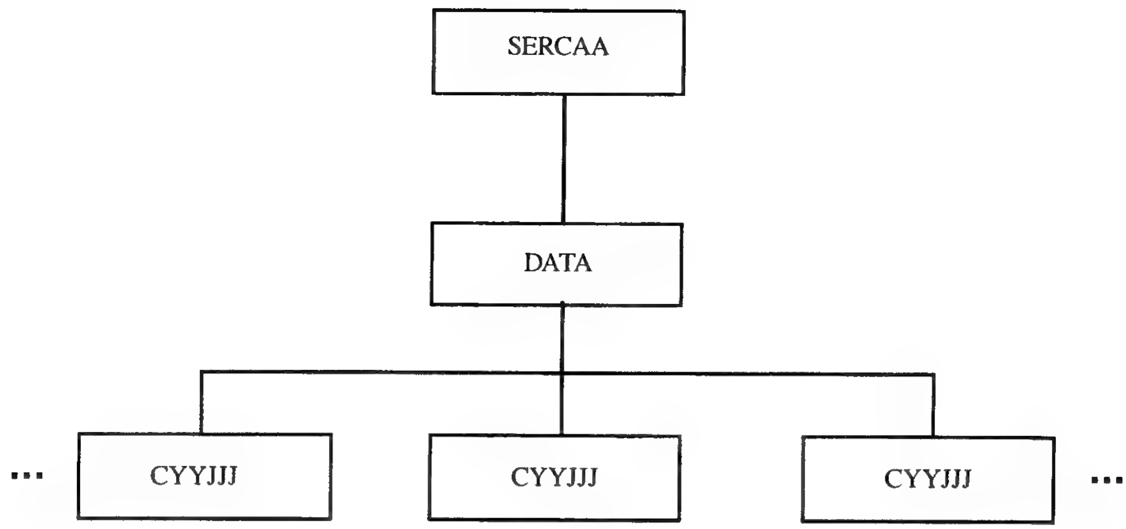

Figure 6. On disk directory structure for SERCAA data. Lowest level directories are named using the date-time of the data in century, year, and Julian day format (CYYJJJ).

data to reconstruct database entries (products excluded) while the naming convention helps to locate and identify data files that are external to the database.

7.0 Database Access From Other Computer Platforms

A computer integration issue existed for SERCAA that involved two dissimilar computer platforms on AIMS that needed to be able to share data in a way that was transparent to the user. Specifically, satellite imagery and products generated and managed on AIMS VAX-based computer platforms needed to be made easily available to users using AIMS SGI imaging workstations for visualization. One solution was to use a component of MultiNET that allows a computer running the UNIX operating system (i.e., an SGI workstation) to mount OpenVMS disks as if they were attached locally. Although image data would be directly accessible, the database that resides on VAX disks would not unless either the database and supporting software were ported or procedures were developed to use the standard UNIX remote-shell program (rsh) to remotely execute existing database applications on a VAX. Porting the database and supporting software would have been a major undertaking not realistically possible under the current program while the use of the remote-shell program would force makeshift techniques to get database information into UNIX-based custom applications.

Instead an alternative solution based on a client-server approach that employs existing database access routines on VAX platforms was chosen. Using the Remote Procedure Call (RPC) protocol developed by Sun MicroSystems (and available on many

UNIX-based computers including SGI), a UNIX-based application that needs to access database information makes a function call similar to the way VAX-based applications do. Input arguments to the call are transferred over the network to a server program running on a VAX which in turn, activates the database module and passes along the input arguments. Once the module has performed the database function, the information is transferred back across the network to the calling program as return arguments. To the UNIX programmer the network aspect of the call is transparent, data are accessed simply by making a function call.

Since MultiNET also provides an RPC object library for the VAX-VMS, testing the RPC mechanism between one of the SGI workstations and a VAX running the MultiNET software was possible. An example application taken directly from the Sun Network Programming Guide was used. Client and server source code were generated on the SGI workstation. The executable client program was then generated directly on the workstation while server source code was moved to the VAX, compiled and then linked. Although the program is simple (it takes a text message from the workstation and prints it on the VAX terminal window), the test proved that the RPC protocol does allow the two platforms to easily share data. Following this simple test, more sophisticated tests were performed to determine if the RPC mechanism could easily handle large satellite image files. A test program to read entire images from the SERCAA database (located on AIMS VAX disks) to an SGI workstation using RPC was developed and test runs were made. On average, images on the order of a megabyte were read over an ethernet-based local-area network that networks all AIMS computers and displayed in a MOTIF window in 5 seconds elapsed time. The results of data load testing on RPC confirmed initial thinking that RPC would be a viable client-server solution for multiplatform access to the SERCAA database.

With encouraging results from the data loading tests, development of a UNIX-based library of data access routines was the next logical step in providing remote access to the SERCAA database. As designed, these routines provide a level of abstraction that hides the details of RPC from the application developer. This means that subroutines and functions with nearly identical calling arguments to the same routines already in use on AIMS are now available to the application developer on UNIX platforms. The initial set of routines supported include:

1. SDB_Query_Core - function that returns entry numbers based on input search criteria.

2. SDB_Get_Data - function that returns the entire contents of an image file or subset based on an input entry number and image bounds.

3. VMS_Mount - function that mounts a VAX-VMS hard disk or magneto-optical drive on behalf of the calling UNIX program.

4. VMS_DisMount - function that dismounts a VAX-VMS hard disk or magneto-optical drive on behalf of the calling UNIX program.

As an example of how an application might incorporate these routines under UNIX, development of a GUI-based application was also undertaken. This program functions as a combination SERCAA database browser and image viewer. The interface is based on MOTIF with callbacks that demonstrate the proper use of the SERCAA database library routines for UNIX. Appendices H through K document the SERCAA database library routines for UNIX.

8.0 Summary

A database management facility was developed to support the development of cloud analysis algorithms under Phase I of the Support of Environmental Requirements for Cloud Analysis and Archive (SERCAA) research and development project. Data requirements under SERCAA Phase I specified processing of sensor data from multiple platforms including DMSP/OLS, NOAA/AVHRR, GOES/VAS, METEOSAT/VISSR, and GMS/VISSR. Sensor data was processed using the highest available spatial resolution for visible and infrared wavelengths and utilized the full range of spectral information available from the satellite sensors. Data was collected from three test-bed areas collectively covering just over half the northern hemisphere from all satellite platforms for the last 10 days of every other month beginning in March 1993. Data collection is expected to end in September of 1995. Multiple 10-day save sets consisting of collected raw data and derived products were maintained online using either fixed-disk mass storage or removable magneto-optical cartridges. The choice of storage medium was based on satisfying the demands of the three processing levels under SERCAA: ingest and registration of raw data, generation of intermediate cloud products, and generation of the final integrated cloud product. Functional requirements for the SERCAA database management facility included the ability to ingest satellite data from multiple platforms and register the data into the database, manage data collected from multiple 10-day data collection periods, manipulate

database entries, and facilitate interactive and programmed access to the database and data files.

Because the functional requirements for the SERCAA database management facility closely paralleled the TACNEPH database effort but on a larger scale, the database architecture under this effort was reviewed and found to be adequate for SERCAA. At the core of the TACNEPH database is a set of interrelated data dictionaries containing information that describes the salient characteristics of image data, latitude-longitude data (used for earth locating image data), satellite-solar geometry data (used to classify solar illuminance conditions and to characterize sun glint in visible reflectance data), and intermediate cloud products. The data dictionaries are organized based on the relational model and are implemented as indexed files.

Deficiencies in the implementation of TACNEPH database management facility were corrected during development of the database management facility for SERCAA. The first problem encountered was that most TACNEPH data dictionary files were fragmented. File fragmentation occurs when a file cannot be located within contiguous blocks on mass storage, resulting in an increase in the amount of time necessary to retrieve file contents. The file fragmentation problem was traced to poor file design. Under SERCAA, file fragmentation was minimized by creating indexed files with characteristics that more closely paralleled files usage as well as performing periodic tuning of the files using VMS utilities. Second, software changes were made to allow users to simultaneously perform read and write operations on data dictionary files. This eliminated a bottleneck under TACNEPH in which database readers were denied access because a data dictionary had been previously opened for exclusive access for write operations. Finally, a pro-active approach was developed to assign, organize and manage records within data dictionaries. Under TACNEPH, new records were added to the database and assigned a number, known as an entry number, to uniquely identify it among other records. To assign an entry number, the TACNEPH logic was to read an entire data dictionary and extract the entry number associated with the last record read. The newly assigned entry number was then computed by incrementing the last entry number used. Although the technique worked it was extremely time consuming. Under SERCAA entry numbers were managed using a file-based bit mask. The bit mask is logically a stream of counted bits, one for each possible entry number up to a maximum of 16K bits. To manage the bit mask, a routine was developed that utilizes VMS built-in functions that directly access hardware and bit manipulation instructions for increased performance.

To accommodate the enormous amount of disk space occupied by data from multiple 10-day save sets within existing disk storage resources on AIMS, the TIFF file

format was adopted for satellite imagery and total cloud analysis products. Using a TIFF software library provided by Silicon Graphics Inc., image data and products were routinely converted to TIFF format as an integral part of the ingest and registration process and during total cloud analysis product generation. When creating a TIFF file, a built-in feature allowed the user to compress the data as it was written out to disk. Using the compression feature, compression ratios of 2 to 1 were commonly achieved for both images and products.

Two new database access routines not previously available under the TACNEPH database management facility were developed for SERCAA. The first was developed to facilitate the use of satellite data within application programs. The general requirement behind the development of the routine was the need to isolate the detail of I/O involved in reading satellite data to a level below the application so that the programmer would simply be presented with the data for an entire image scene or a subset of the scene. Specific requirements were driven by the different ways users wanted to access one or more satellite images. These included access to the entire image at one time, random access to multiple scan lines of the image, and access to subsets within an image. Based on these requirements the routine, known as SDB_GET_DATA, was developed and incorporated into the SERCAA database software library and subsequently used by total cloud analysis programs.

The second routine was developed to allow database application developers to query the database for entries based on image date-time, satellite identifier, and test bed location. In order to support queries based on these parameters, it was necessary to generate a new data dictionary containing these parameters (plus the entry number) in sorted order. Once sorted a user would call the new routine from a program; which in turn, performed a binary search on the sorted list to quickly locate the entry numbers associated with the query parameters specified as calling arguments to the routine. As implemented, the new data dictionary was maintained alongside the other data dictionaries. When entries were either added to or deleted from the core data dictionary, a process would automatically be triggered to update the sorted data dictionary to reflect the changes. With a sorted data dictionary in place, the new routine was then developed. The routine, known as SDB_QUERY_CORE, was incorporated into the SERCAA database software library and subsequently used by application programs. It was also incorporated into a general-purpose listing utility known as SDB. With SDB, the user was able to query the database for data dictionary information based on the query parameters supported by SDB_QUERY_CORE and have a listing formatted to the display screen or to a file. SDB could also be used to display data dictionary information based on known entry numbers.

Satellite data ingest and database registration procedures used early on in SERCAA were difficult to use, required a high level of operator intervention, demanded code changes if data were targeted for different storage areas, and did not support manual override actions in cases where they were necessary (e.g., computer failure). To address these deficiencies, new goals were established to streamline ingest and database registration procedures for all satellite platforms so that satellite data could quickly and efficiently be made available to cloud analysis applications in a timely manner. A procedural approach common to all satellite platforms in which the ingest and registration of satellite data is defined as a series of well defined steps was developed for the streamlining task. During normal operations these steps would be automated but in the case of failure during any of the steps, an operator was able to resume automated operations beginning at the step at which the failure had occurred. To allow the operator to respond to changing storage conditions on AIMS, an editable file was established. With this file the operator was able to specify the source location of satellite data, any intermediate location needed to store files, and the final storage location known by the database. This file also contained other parameters that could change periodically including the choice of computer to perform database registration and identification of the 10-day set (month and year). The streamlining task also introduced a new file known as the SDB file. This file contained all the information necessary to register satellite and ancillary data and was also used to record the entry numbers assigned to the registered files by the database management facility. Because the SDB file contained an audit trail of information covering the entire registration procedure, it became an important file for reference during instances in which registration had failed and required manual intervention.

A file and directory naming convention was adopted as part of the streamlining process to allow the user to easily identify and locate files should the database become corrupt or otherwise unreadable. File names were generated using files characteristics (data type, area of coverage, and date-time of data) while directories were named using the year and Julian day. During registration the appropriate directory to be used to store files was identified by examining the valid date-time of the data. If the directory did not exist, it was automatically created.

Finally, a computer integration issue involving two dissimilar AIMS computer platforms that needed to share data in a way that was transparent to the user was solved using a client-server approach. At issue was the fact that satellite imagery and products as well as database files generated and managed with the SERCAA database management facility on AIMS VAX computers (running the VMS operating system) could not be accessed directly by users using AIMS SGI imaging workstations (running the UNIX

operating system) to query and subsequently view the data. Of the solutions considered, only one, the Remote Procedure Call (RPC) protocol developed by SUN Microsystems, supported transparent access to image and database files without having to port database software to the SGI workstations. With RPC, a UNIX-based application program that needs to access imagery and database information makes function calls similar to the way VAX-based applications do. Input arguments to the call are transferred over the network to a server program running on a VAX which, in turn, activates the appropriate database module and passes along the input arguments. Once the module has performed the function, any return information is transferred back across the network to the calling program as return arguments. On AIMS, an RPC software library was already included with the SGI workstations when they were purchased. An RPC library for VAX computers was provided by MultiNET, a networking software product for VAX systems that links VAX and other computers that support the TCP/IP networking family of protocols.

To support the equivalent database functionality already established on AIMS VAX computers, a software library of database access routines based on RPC was developed for the SGI imaging workstations. The routines provided a level of abstraction that hid the details of RPC from the application developer so that the routines were almost identical to their VAX-based counterparts. The set of database access routines developed included the database query routine (SDB_Query_Core), the image access routine (SDB_Get_Data), and the database primitives.

9.0 References

Gustafson, G. and R. d'Entremont, 1992: Single channel and multispectral cloud algorithm development for TACNEPH. *Proceedings, Sixth Conference on Satellite Meteorology and Oceanography.* Atlanta, GA, American Meteorology Society, Boston, MA, pp. 13-16.

Gustafson, G.B., R.G. Isaacs, R.P. d'Entremont, J.M. Sparrow, T.M. Hamill, C. Grassotti, D.W. Johnson, C.P. Sarkisian, D.C. Peduzzi, B.T. Pearson, V.D. Jakabhazy, J.S. Belfiore, and A.S. Lisa, 1994a: Support of Environmental Requirements for Cloud Analysis and Archive (SERCAA). PL-TR-94-2114, Phillips Laboratory, Hanscom AFB, MA, 105 pp. ADA283240

Gustafson, G.B., R.G. Isaacs, J.M. Sparrow, D.C. Peduzzi, and J.S. Belfiore, 1994b: Automated Satellite Cloud Analysis - Tactical Nephanalysis (TACNEPH). PL-TR-94-2160, Phillips Laboratory, Hanscom AFB, MA, 100pp.

Hamill, T.M. and R.N. Hoffman, 1993: SERCAA Cloud Analysis and Integration; Design Concepts and Interaction with Cloud Forecast Models. PL-TR-93-2100. Phillips Laboratory, Hanscom AFB, MA 01731. **ADA269104**

Ivaldi, C.F., G.B. Gustafson, and J. Doherty, 1992: The Air Force Interactive Meteorological System: A Research Tool For Satellite Meteorology. PL-TR-92-2327, Phillips Laboratory, Hanscom AFB, MA, 85pp. ADA265957

Shell, D.L., 1959: A high-speed sorting procedure, *Comm. ACM*, **14**, 11, pp. 30-32.

Welch, T.A., 1984: A technique for high performance data compression., *IEEE Computer*, **17**,6.

SDB_BitMask

SDB_Bitmask is an Integer*4 function used to manage a file-based bit mask.

Format Istatus = SDB_Bitmask(Function, Entries_Array, Number_Entries)

Returns Status return values are listed under Condition Values Returned.

Arguments:

Function is an **Integer*4** that indicates what function to perform. Supported functions include:

Value	Meaning
1	Allocate entries
2	Deallocate entries
3	Allocate entries from input (used to re/construct a bit mask)
4	Provide a list of allocated entries (used to reconstruct a dictionary)
5	Return the current total number of allocated entries

Entries_Array is an **Integer*4** array reference dimensioned internally by the argument **Number_Entries**. Its meaning is goverened by the input argument **Function**.

Function	Argument Type	Meaning
2	Input	Contains the entries to be deallocated
3	Input	Contains the entries to be used to re/construct the bit mask
1	Output	Contains allocated entries
4	Output	Contains a list of all allocated entries

Number_Entries is an **Integer*4**. Its meaning is governed by the input argument **Function**.

Function	Argument Type	Meaning
1	Input	Number of entries to allocate
2	Input	Number of entries in Entries_Array to deallocate
3	Input	Number of entries in Entries_Array to use for bit mask re/construction
4	Input	Dimension of Entries_Array
4,5	Output	Total number allocated entries in the bit mask

Description:

SDB_Bitmask is a multi-function routine used to manage a file-based bit mask. The bit mask is used in conjunction with the SERCAA database core data dictionary. Functions allow the user to allocate and deallocate entries and to retrieve the current total number of entries in the core data dictionary. Additionally, the routine can be used to assist in the reconstruction of a corrupted core data dictionary by providing a list of known allocated entry numbers that can be used in conjunction with secondary data dictionaries to cross-reference internal entries. In the case of a corrupted or otherwise unusable bit mask file, the routine can also be used to reconstruct the bit mask from entries stored in a core data dictionary.

Condition Values Returned:

SS$_Normal -- Normal completion status value.

BIT__No_Avail_Entries -- No entries available for allocation.

BIT__Invalid_Entry -- One or more of the input entries is invalid.

BIT__Invalid_Function -- Invalid function requested.

BIT__Array_Too_Small -- The size of the input array is too small.

Appendix B: SERCAA Database Primitives

The SERCAA database primitives are the lowest-level routines in the database software hierarchy. These routines directly access the on-disk index file structure of the data dictionaries using FORTRAN I/O.

SDB_Get_Core_DD - Subroutine that retrieves the contents of a core data dictionary entry based on an input entry number.

SDB_Get_SatImg_DD - Subroutine that retrieves the contents of an image data dictionary entry based on an input entry number.

SDB_Get_LatLon_DD - Subroutine that retrieves the contents of a latitude-longitude data dictionary entry based on an input entry number.

SDB_Get_Angles_DD - Subroutine that retrieves the contents of an angles data dictionary entry based on an input entry number.

SDB_Get_Product_DD - Subroutine that retrieves the contents of a product data dictionary entry based on an input entry number.

SDB_Add_Core_Entry - Subroutine that adds an entry to the core data dictionary. Input is a structure containing the contents of the entry.

SDB_Add_SatImg_Entry - Subroutine that adds an entry to the image data dictionary. Input is a structure containing the contents of the entry.

SDB_Add_LatLon_Entry - Subroutine that adds an entry to the latitude-longitude data dictionary. Input is a structure containing the contents of the entry.

SDB_Add_Angles_Entry - Subroutine that adds an entry to the angles data dictionary. Input is a structure containing the contents of the entry.

SDB_Add_Product_Entry - Subroutine that adds an entry to the product data dictionary. Input is a structure containing the contents of the entry. New product entries are added starting at the computed base entry and increase sequentially. The entry number within the input structure must be a valid core data dictionary product entry number.

SDB_Update_Core_Entry - Subroutine that modifies an existing entry in the core data dictionary. Input is a structure containing the modified contents of the entry.

SDB_Update_SatImg_Entry - Subroutine that modifies an existing entry in the image data dictionary. Input is a structure containing the modified contents of the entry.

SDB_Update_LatLon_Entry - Subroutine that modifies an existing entry in the latitude-longitude data dictionary. Input is a structure containing the modified contents of the entry.

SDB_Update_Angles_Entry - Subroutine that modifies an existing entry in the angles data dictionary. Input is a structure containing the modified contents of the entry.

SDB_Update_Product_Entry - Subroutine that modifies an existing entry in the product data dictionary. Input is a structure containing the modified contents of the entry.

The entry number within the input structure must be a valid product data dictionary entry number.

SDB_Delete_Core_Entry - Subroutine that deletes a core data dictionary entry. Input is the entry number of the entry to delete. Optionally deletes the data file associated with the entry. Also clears the bit within the bit mask that corresponds to the entry number.

SDB_Delete_SatImg_Entry - Subroutine that deletes a data dictionary entry from the image data dictionary. Input is the entry number of the entry to delete.

SDB_Delete_LatLon_Entry - Subroutine that deletes a data dictionary entry from the latitude-longitude data dictionary. Input is the entry number of the entry to delete.

SDB_Delete_Angles_Entry - Subroutine that deletes a data dictionary entry from the angles data dictionary. Input is the entry number of the entry to delete.

SDB_Delete_Product_Entry - Subroutine that deletes a data dictionary entry from the product data dictionary. Input is the entry number of the entry to delete. Entry number must be a valid product data dictionary entry number.

Appendix C: SERCAA Database Data Access Routine

SDB_Get_Data

SDB_Get_Data is an unsigned integer*4 function used to retrieve satellite image data from the SERCAA database.

Format	SDB_Get_Data	entry, min_sample, max_sample,min_line,max_line, array

Returns Status return values are listed under Condition Values Returned.

Arguments:

entry

usage: word_signed
type: word integer (signed)
access: read only
mechanism: by reference

entry is the database entry number associated with the data to be retrieved.

min_sample

usage: word_signed
type: word integer (signed)
access: read/write
mechanism: by reference

min_sample is the minimum x area coordinate (one relative) within the image to begin the retrieval. An input value of zero instructs the routine to use the coordinate from the database.

max_sample

usage: word_signed
type: word integer (signed)
access: read/write
mechanism: by reference

max_sample is the maximum x area coordinate (one relative) within the image to end the retrieval. An input value of zero instructs the routine to use the coordinate from the database.

min_line

usage: word_signed

type: word integer (signed)
access: read/write
mechanism: by reference

min_line is the minimum y area coordinate (one relative) within the image to begin the retrieval. An input value of zero instructs the routine to use the coordinate from the database.

max_line

usage: word_signed
type: word integer (signed)
access: read/write
mechanism: by reference

max_line is the maximum y area coordinate (one relative) within the image to begin the retrieval. An input value of zero instructs the routine to use the coordinate from the database.

array

usage: word_signed
type: word integer (signed)
access: read/write
mechanism: by reference

As a read only variable, **array** contains a memory address to preallocated memory. Upon return, the memory area referenced by **array** will contain the data. As a write only variable, **array** contains a value of zero and upon return will contain a memory address that references the location of the data.

Description:

SDB_Get_Data is an integer function that retrieves satellite image data from the SERCAA database. Input parameters specify starting offsets and the number of lines and samples to take starting at the offsets. Zero offsets imply the user is requesting the entire image. If the specified image subsection partially exceeds the image bounds, the user's buffer will be zero-filled as needed on a line-by-line basis. The offsets and dimensions upon return will reflect the area within the user buffer that contains valid data. The function variable will also contain a status value indicating a change in dimensions of valid data. The buffer area for data is allocated either by the user or by the routine. SDB_Get_Data recognizes and handles raw and TIFF file formats.

Restrictions apply to requests that define a subset (or block) of the image. First, the initial call to SDB_Get_Data regulates the block size; future calls cannot deviate from the initial block size. Secondly, block access is sequential. Direct access to records within the file is not supported.

SDB_Get_Data uses a linked list to manage requests. The list uses entry number and the buffer area to arbitrate separate requests. This allows the user to retrieve data from multiple image files within the same program or to make multiple requests to the same image file with different block sizes.

SDB_Get_Data makes extensive use of dynamic memory. It is important that this memory be freed once the data returned by SDB_Get_Data is no longer needed so that the calling process can reuse the virtual memory. This is done by calling the function

SDB_Free_Data. In addition to freeing memory, this routine also closes the image file and removes the associated entry from the linked list.

Example Code Stub in C:

```
#include "sdb_includes:sdb_access.h"
#include "sdb_includes:sdb_errors.h"

main()
{

        short           entry;
        short           min_sample;
        short           max_sample;
        short           min_line;
        short           max_line;
        int             array = 0;
        unsigned int    status;
        unsigned char   *data;
        int             i;
        int             total_data;

/* Fill in input arguments */

        entry = 900;
        min_sample = 0;
        max_sample = 0;
        min_line = 0;
        max_line = 0;

/* Get the data */

        status = sdb_get_data(&entrry, &min_sample, &max_sample,
                              &min_line, &max_line, &array);
        if (status != PASS) {
         printf("Failure on call to SDB_Get_Data\n");
         exit(1);
        }

        data = (unsigned char *) array;
        total_data = max_sample * max_line;

        for (i=0; i<total_data; i++) {
         printf("%d\n",data[i]);
        }

/* Free resources */

        status = sdb_free_data(&entry, &array);
        if (status == FAIL) {
         printf("Failure on call to SDB_Free_Data\n");
        }
    }
```

Example Code Stub in FORTRAN:

```
        Program Example
C
        Include 'SDB_Includes:SDBA.Inc'
        Include 'SDB_Includes:SDB_Errors.Inc'
        Integer*2 Entry,
   &              Min_Sample,
   &              Max_Sample,
   &              Min_Line,
   &              Max_Line
        Integer*4 Status,
   &              Array        /0/,
                  Total_Data
        Character*60 Message
C
C  Fill in input arguments
C
        Entry = 900
        Min_Sample = 0
        Max_Sample = 0
        Min_Line = 0
        Max_Line = 0
C
C  Get the data
C
        Status = SDB_Get_Data(Entry, Min_Sample, Max_Sample,
                              Min_Line, Max_Line, Array)
        If (Status .NE. PASS) Then
             Call SDB_Get_Error(Status, Message)
             Print *, Message
        Call Exit
        EndIf
C
C  To get at the data, call a routine and de-reference the address returned
C  from SDB_Get_Data
C
        Total_Data = Max_Sample * Max_Line
        Call Get_Data(%Val(Array), Total_Data)
C
C  All done so free dynamic virtual memory
C
        Status = SDB_Free_Data(Entry, Array)
        Call Exit
        End
        Integer*4 Function Get_Entries(Data, Total_Data)
        Integer*4 Total_Data
        Byte Data(Total_Data)
        Do I = 1, Total_Data
         Print *, Data(I)
        EndDo
        Return
        End
```

Condition Values Returned:

PASS -- Normal completion status value.

SDB_CHGDIM -- Sample and/or line dimensions have changed. Check the return values.

SDB_ITEMNOTFOUND -- An item could not be found in the linked list for this call.

SDB_ALLOCFAIL -- An attempt to dynamically allocate virtual memory failed.

SDB_NOTINDATA -- The data request was out of range.

SDB_BADFT -- The file type is invalid.

SDB_FOERR -- The file could not be opened.

SDB_BADSTRIP -- Strip was out of sequence.

SDB_MINGTMAX -- Minimum sample/line was greater than maximum sample/line.

SDB_READERR -- A "C" I/O file read error was encountered.

SDB_Query_Core

SDB_Query_Core is an integer*4 function used to query the SERCAA core data dictionaries for entries that satisfy input query criteria.

Format SDB_Query_Core function, query, address, number_entries

---

Returns Status return values are listed under Condition Values Returned.

Arguments:

function

VMS usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by value

function controls the function performed by SDB_Query_Core. The two functions are 1) perform a query on the core data dictionary or 2) deallocate dynamic memory used to accomplish 1). The **function** argument is a longword value containing one of two pre-defined symbolic constants. The constants are:

SDB__Query_Core
SDB__Free_Memory

query

VMS usage: structure
type: structure
access: read only
mechanism: by reference

query is a structure containing the query parameters used to perform the search for core entries that satisfy the query criteria. The elements of this structure are:

start_yyjjjhhmm is a signed longword containing the start date-time of the query in YYJJJHHMM format or 0 (NULL) to indicate a wild card operation.

stop_yyjjjhhmm is a signed longword containing the stop date-time of the query in YYJJJHHMM format or 0 to indicate a wild card operation. If not 0, must be equal to greater than **start_yyjjjhhmm.**

satellite is a signed longword containing the satellite code as defined in the SERCAA database or 0 for a wild card operation.

location is a signed longword containing the location code or 0 for a wild card operation. **location** is specified by the use of symbolic constants. They are:

SDB__SAT
SDB__SDT
SDB__SET

match_flag is a signed longword containing the match criteria to use for the starting date-time. **match_flag** is specified by the use of symbolic constants. They are:

EXACT	attempt an exact match on the start date-time specified in **start_yyjjjhhmm.** ***(Default)***
CLOSEST	find the closest match to **start_yyjjjhhmm.**
CLOSEST_NOOVER	find the closest match to **start_yyjjjhhmm** without exceeding it.

core_flag is a signed longword used to indicate whether to provide memory references to core entry structures in addition to entry numbers. Possible values are:

TRUE	provide memory references to entry structures.
FALSE	don't provide memory references. ***(Default)***

address

VMS usage: address
type: longword (unsigned)
access: write only
mechanism: by reference

address is an address returned by SDB_Query_Core that references an array of structures of type "entries" (defined in include files). The structure is composed of the following elements:

entry_number is a signed longword containing a core data dictionary entry number that satisfies the query criteria.

entry_address is an unsigned longword containing the address of the core data dictionary entry structure that corresponds to **entry_number.** The contents of **entry_address** are valid only if **core_flag** (**query** structure) is set to TRUE.

number_entries

VMS usage: longword_signed
type: longword integer (signed)
access: write only
mechanism: by reference

number_entries is the total number of entries found that satisfied the query criteria. **number_entries** also indicates the size of the array of structures "entries" defined above. **number_entries** can range from 0 (no entries satisfied the query criteria) to the total number of entries in the core data dictionary.

Description:

SDB_Query_Core is a programming interface to sorted core data dictionaries maintained in parallel with the keyed core data dictionaries used for SERCAA. The sorted core data dictionaries are triply sorted on the parameters defined in the QUERY structure used in this function, namely: date-time, satellite id and location id. This function is designed to retrieve entry numbers that satisfy the query criteria setup by the programmer in the QUERY structure. Note that any combination of specified and non-specified (i.e. wild card operation) query parameters is possible. If all parameters are wild carded, all entry numbers will be returned. In addition to returning entry_numbers, SDB_Query_Core can also return memory references to the actual entry structures which are maintained internal to the function. This is particularly useful for applications that may want to make multiple calls to this function and reference fields of the core entries.

SDB_Query_Core makes extensive use of dynamic memory. It is important that this memory be freed once the data returned by SDB_Query_Core is no longer needed so that the calling process can reuse the virtual memory. This is done by calling the function with **function** set to SDB__Free_Memory. The following table summarizes when to make this call:

Scenario of function call	When to free dynamic virtual memory
Single call to the function, return entry numbers only	Immediately following the call
Multiple calls to the function, return entry numbers only	After the last call
Single or multiple calls to the function, entry numbers and entry structures	When entry structures are no longer referenced

Example Code Stub in FORTRAN:

```
        Program Example
C
        Include 'SDB_Includes:SDB_Query_Core.Inc'
        Include 'SDB_Includes:SDB_Errors.Inc'
        Include 'SDB_Includes:SDBA.Inc'
C
        Record /Query_Structure/ Query
        Integer*4 Fun,
  &                Address,
  &                Num_Entries,
  &                Status,
  &                I
        Character*60 Message
C
C
C
C  Fill in query structure.  Request is for GOES-7 data on 93151 at 1230Z.
C
        Fun = SDB__Query_Core
        Query.Start_YYJJJHHMM = 931511230
        Query.Stop_YYJJJHHMM = 931511230
        Query.Satellite = GOES_7
        Query.Location = SDB__SAT
        Query.Match_Flag = EXACT
        Query.Core_Flag = .False.
C
C  Get entry numbers that satisfy query criteria parameters
C
        Status = SDB_Query_Core(%Val(Fun), Query, Address, Num_Entries)
        If (Status .NE. PASS) Then
                Call SDB_Get_Error(Status, Message)
                Print *, Message
                Call Exit
        EndIf
C
C  To get at output structure, call a routine and de-reference the address returned
C  from SDB_Query_Core
C
        Status = Get_Entries(%Val(Address), Num_Entries)
        If (Status .NE. PASS) Then
                < process status >
        EndIf
C
C  All done so free dynamic virtual memory
C
        Fun = SDB__Free_Memory
        Status = SDB_Query_Core(%Val(Fun), Query, Entries, Num_Entries)
C
        Call Exit
        End
```

```
      Integer*4 Function Get_Entries(Entry_Info, Num_Entries)
C
      Include 'SDB_Includes:SDB_Query_Core.Inc'
      Include 'SDB_Includes:SDBA.Inc'
      Include 'SDB_Includes:Core_DD.Inc'
      Include 'SDB_Includes:Satimg_DD.Inc'
      Record /Entry_Info_Structure/ Entry_Info(Num_Entries)
      Record /Core_DD/ Core_DD
      Record /Satimg_DD/ Satimg_DD
      Integer*4 Num_Entries,
     &          Image_Buffer_Address     /0/,
     &          SDB_Get_Data
C
C Assume success
C
      Get_Entries = PASS
C
C Once inside this routine we can get at the entry numbers by simply accessing
C the member 'ENTRY_NUMBER' of the structure 'ENTRY_INFO'.  For example:
C
      Do I = 1, Num_Entries
          Write (6,*) 'Entry numbers are', Entry_Info(I).Entry_Number
      EndDo
C
C Or to get at the actual data (a little more work)
C
      Do I = 1, Num_Entries
          Call SDB_Get_Core_DD(Entry_Info(I).Entry_Number, Core_DD, Status)
          If (Status .NE. 0) Then
              < process status >
          EndIf
          If (Core_DD.Data_Type .EQ. GR_IMAGE) Then
              Call SDB_Get_Satimg_DD(Entry_Info(I).Entry_Number,
                                     Satimg_DD,
                                     Status)
              If (Status .NE. 0) Then
                  < process status >
              EndIf
              Status = SDB_Get_Data(Entry_Info(I).Entry_Number,
     &                              1,
     &                              Satimg_DD.Elem_Per_Line,
     &                              1,
     &                              Satimg_DD.Num_Lines,
     &                              Image_Buffer_Address)
              If (Status .NE. PASS) Then
                  < process status >
              EndIf
              .
              < process data >
              .
          EndIf
      EndDo
      Return
      End
```

Example Code Stub in C:

```
#include <stdlib.h>
#include "sdb_includes:sdb_query_core.h"
#include "sdb_includes:sdb_access.h"
#include "sdb_includes:core_dd.h"
#include "sdb_includes:satimg_dd.h"
#include "sdb_includes:sdba.h"

main()
{
        QUERY           query;
        ENTRIES         *entry_info;
        CORE            *core_dd;
        SATIMG          satimg_dd;

        int             fun;
        int             entries;
        int             num_entries;
        int             status;
        int             i;
        short int       min_sample = 1;
        short int       min_line = 1;
        int             image_buffer_address = 0;

/*  Fill in query structure.  Request is all NOAA-11 data for the case day 93145.  */

        fun = SDB__Query_Core;
        query.start_yyjjjhhmm = 931450000;
        query.stop_yyjjjhhmm = 931452359;
        query.satellite = NOAA_11;
        query.location = 0;
        query.match_flag = CLOSEST;
        query.core_flag = TRUE;

/*  Get entry numbers that satisfy query criteria parameters  */

        status = sdb_query_core(fun, &query, &entries, &num_entries);

/*  Type cast the return address to the typedef'd structure of type 'ENTRIES'  */

        entry_info = (ENTRIES *) entries;
```

```
/* Once we type cast, we can access the members of the first structure in the array
   of structures.  The members are:
        entry_info->entry_number
        entry_info->entry_address

   Loop thru number of found entries.  Each time thru the loop type cast
   entry_info->entry_address so that it can be accessed as a core entry structure.
   At the bottom of the loop advance to the next structure in the array of structures  */

        for (i=0; i<num_entries; i++) {

                core_dd = (CORE *) entry_info->entry_address;

                printf("\n Core entry is %d \n", core_dd->entry);
                printf("\n Core data type is %d \n", core_dd->data_type);
                printf("\n Core satellite code is %d \n", core_dd->sat_code);
                printf("\n Core sensor code is %d \n", core_dd->sensor_code);
                printf("\n Core country code is %s \n", core_dd->country_code);
                printf("\n Core data file is %s \n", core_dd->data_file);

                if (core_dd->data_type == GR_IMAGE) {
                        sdb_get_satimg_dd(&core_dd->entry, &satimg_dd, &status);
                        if (status != 0) {
                                < process status >
                        }
                        status = sdb_get_data(&core_dd->entry,
                                              &min_sample,
                                              &satimg_dd.elem_per_line,
                                              &min_line,
                                              &satimg_dd.num_lines,
                                              &image_buffer_address);
                        If (status != PASS) {
                                < process status >
                        }
                        .
                        < process data >
                        .
                }
                entry_info++;
        }

/* All done so free dynamic virtual memory  */

        fun = SDB__Free_Memory;
        status = sdb_query_core(fun, &query, &entries, &num_entries);
}
```

Condition Values Returned:

PASS -- Normal completion status value.

SDB$FOERR -- A file open error occurred on the sorted core data dictionary file.

SDB$READERR -- A read error occurred on the sorted core data dictionary file.

SDB$ALLOCFAIL -- An attempt to dynamically allocate virtual memory failed.

SDB$COREACCESSFAIL -- An error occurred while processing data from the keyed core data dictionary file.

SDB$DTCALCFAIL -- An error occurred while making a date-time calculation. This is usually the result of a corrupt date-time within a data dictionary.

SDB$DTOUTOFRNG -- The start and/or stop date-time parameter(s) in the QUERY structure are/is out of the range of the data dictionary.

Appendix E: Interactive SERCAA Database Utility (SDB)

The SDB application provides users of the SERCAA database with a flexible tool to browse database dictionary entries. The SDB command allows users to narrow their search for specific data by accepting qualifiers which correspond to specific parameters. Abbreviations may be used when specifying all qualifiers.

FORMAT: SDB[/Qualifiers[=Parameter]]

QUALIFIERS:

/QUERY=CORE or /CORE

The /QUERY=CORE qualifier allows the user to examine the core data dictionary. All listed search qualifiers are supported for this dictionary. /CORE was maintained to provide consistency with the previous version of SDB.

EXCEPTIONS:

/MOD is not supported
/DISKS is not supported
/DAYS is not supported

/QUERY=IMAGE or /IMAGE

The /QUERY=IMAGE qualifier allows the user to examine the image data dictionary. All listed search qualifiers are supported for this dictionary. /IMAGE was maintained to provide consistency with the previous version of SDB.

EXCEPTIONS:

/DISKS is not supported
/DAYS is not supported

/QUERY=LATLON or /LATLON

The /QUERY=LATLON qualifier allows the user to examine the latlon data dictionary. All listed search qualifiers are supported for this dictionary. /LATLON was maintained to provide consistency with the previous version of SDB.

EXCEPTIONS:

/DISKS is not supported
/DAYS is not supported

/QUERY=ANGLES or /ANGLES

The /QUERY=ANGLES qualifier allows the user to examine the angles data dictionary. All listed search qualifiers are supported for this dictionary. /ANGLES was maintained to provide consistency with the previous version of SDB.

EXCEPTIONS:

/DISKS is not supported
/DAYS is not supported

/QUERY=PRODUCT or /PRODUCT

The /QUERY=PRODUCT qualifier allows the user to examine the product data dictionary. All listed search qualifiers are supported for this dictionary. /PRODUCT was maintained to provide consistency with the previous version of SDB.

EXCEPTIONS:

/DISKS is not supported
/DAYS is not supported

/QUERY=(CORE,IMAGE)

The /QUERY=(CORE,IMAGE) qualifier is the default for SDB. This qualifier allows the user to examine entries in the core and image data dictionaries. The displayed information is a combination of data from both data dictionaries formatted on a single line of output. If output is directed to the screen, scroll control is used. After 20 lines of output are displayed to the screen, the user is prompted to continue or terminate output. When the return key is depressed output continues preceded by re-display of the header. Scroll control is suppressed when output is directed to a file.

EXCEPTIONS:

/MOD is not supported
/DISKS is not supported
/DAYS is not supported

/QUERY=(PRODUCT,IMAGE)

The /QUERY=(PRODUCT,IMAGE) qualifier allows the user to examine entries that are registered in the image and product data dictionaries. The displayed information is a combination of data from both data dictionaries plus the image file location. If output is directed to the screen, scroll control is used. After 20 lines of output are displayed to the screen, the user is prompted to continue or terminate output. When the return key is depressed output continues preceded by re-display of the header. Scroll control is suppressed when output is directed to a file.

EXCEPTIONS:

/MOD is not supported
/DISKS is not supported
/DAYS is not supported

/QUERY=(LATLON, ANGLES)

The /QUERY=(LATLON,ANGLES) qualifier allows the user to examine the entry numbers that are registered in the latlon and angles data dictionaries. The displayed information is a combination of data from both data dictionaries. If output is directed to the screen, scroll control is used. After 20 lines of output are displayed to the screen, the user is prompted to continue or terminate output. When the return key is depressed output continues preceded by re-display of the header. Scroll control is suppressed when output is directed to a file.

EXCEPTIONS:

/MOD is not supported
/DISKS is not supported
/DAYS is not supported

/CLEAR

The /CLEAR qualifier erases the screen of the terminal prior to displaying the requested data. Users must also indicate a specific database.

Example:

$SDB/IMAGE/MOD/CLEAR

will clear the terminal screen and then specify modification dates of all registered image database entries.

/MOD

The /MOD qualifier displays the modification date of registered entries. Users must also indicate a specific dictionary. /CORE and /QUERY=CORE display the modification date by default.

Example:

$SDB/IMAGE/MOD

will display modification dates of the image dictionary entries.

UNSUPPORTED DICTIONARIES:
/QUERY= CORE or /CORE
/QUERY=(CORE,IMAGE)
/QUERY=(PRODUCT,IMAGE)
/QUERY=(LATLON,ANGLES)

/FULL

The /FULL qualifier allows the user to examine an entire structure for a specified entry number. /FULL MUST be used with /ENTRY. Users must also indicate a specific dictionary. /FULL may be used with the following qualifiers:

/IMAGE
/QUERY=CORE
/QUERY=IMAGE
/QUERY=LATLON
/QUERY=ANGLES
/QUERY=PRODUCT

Example:

$SDB/IMAGE/FULL/ENTRY=2

will display all information of the image structure for entry 2.

UNSUPPORTED COMBINATIONS:

/OUT is not currently supported with /FULL

/LOCATION

The /LOCATION qualifier allows the user to specify the geographic area of interest. Users must also choose a specific dictionary from the list of qualifiers. Valid keywords are SAT, SET or SDT.

Example:

$SDB/CORE/LOCATION=SAT or
$SDB/CORE/LOC=SAT

will display all registered entries in the core dictionary for the location SAT.

UNSUPPORTED COMBINATIONS:

/BEFORE AND /AFTER is not supported with /LOC

/SAT

The /SAT qualifier allows the user to specify the platform of interest. Users must choose a specific dictionary. Valid platform keywords are as follows:

Data are available for:
DMSP_F10, DMSP_F11, NOAA_11, NOAA_12, GOES_7, GMS and MET_5

Other valid keywords:
MET_3, MET_4, MET_6, DMSP_F8, DMSP_F9, TIRSO_N, NOAA_6, NOAA_7, NOAA_8, NOAA_9, NOAA_10 and GOES_NEXT

Example:

$SDB/QUERY=LATLON/SAT=GMS

will display all entries in the latlon dictionary for the platform GMS.

/TO

The /TO qualifier allows the user to specify a time range of interest. /TO must be used with /FROM, where /FROM indicates the beginning time and /TO indicates the ending time. Users must also indicate a specific dictionary. /TO and /FROM may be used with all dictionaries. Times may be given in the following formats:

YYDDD - to specify a beginning and ending date
YYDDDHH - to specify a beginning and ending hour
YYDDDHHMM - to specify a beginning and ending minute

Users are constrained to use the same format for the /TO and /FROM qualifiers. Thus, if you specify to the hour with the /FROM qualifier you must also specify to the hour with the /TO qualifier.

Example:

$SDB/TO=9314405/FROM=9314407/ANGLES

will display all entries in the angles dictionary between the hours 05 and 07 on Julian day 93144.

UNSUPPORTED COMBINATIONS:

/TIME is not supported with /TO and /FROM
/CASE is not supported with /TO and /FROM

/FROM

The /FROM qualifier allows the user to specify a time range of interest. /FROM must be used with /TO, where /TO indicates the ending time and /FROM indicates the beginning time. Users must also indicate a specific dictionary. /FROM and /TO may be used with all dictionaries. Times may be given in the following formats:

YYDDD - to specify a beginning and ending date
YYDDDHH - to specify a beginning and ending hour
YYDDDHHMM - to specify a beginning and ending minute

Users are constrained to use the same format for the /FROM and /TO qualifiers. Thus, if you specify to the hour with the /FROM qualifier you must also specify to the hour with the /TO qualifier.

Example:

$SDB/FROM=9314407/TO=9314405/ANGLES

will display all entries in the angles dictionary between the hours 05 and 07 on Julian day 93144.

UNSUPPORTED COMBINATIONS:

/TIME is not supported with /TO and /FROM
/CASE is not supported with /TO and /FROM

/TIME

The /TIME qualifier allows the user to specify a date or time of interest. If no entries exist for the specified time, SDB will return the entries that are closest to the indicated time. Users must also indicate a specific dictionary. /TIME may be used with all dictionaries. Times may be given in the following formats:

YYDDD - to specify a date
YYDDDHH - to specify an hour
YYDDDHHMM - to specify a minute

Example:

$SDB/TIME=9314405/PRODUCT

will display all entries in the product dictionary for the hour 05 on Julian day 93144.

UNSUPPORTED COMBINATIONS:

/TIME is not supported with /TO and /FROM
/TIME is not supported with /CASE

/CASE

The /CASE qualifier allows the user to specify a date of interest. Users must also indicate a specific dictionary. /CASE may be used with all dictionaries. Times must be given in the following format:

YYDDD - to specify a date

Example:

$SDB/CASE=93144/QUERY=CORE

will display all entries in the core dictionary for Julian day 93144.

UNSUPPORTED COMBINATIONS:

/CASE is not supported with /TO and /FROM
/CASE is not supported with /TIME

/ENTRY

The /ENTRY qualifier allows the user to specify a specific entry number. Users must also indicate a specific dictionary. /ENTRY may be used with all dictionaries. If entry number is used for the product dictionary, the user must specify the product entry number from the core dictionary.

Example:

```
$SDB/ENTRY=5/QUERY=IMAGE
```

will display entry 5 from the image dictionary.

UNSUPPORTED COMBINATIONS:

/ENTRY is not supported with /QUERY=(LATLON,ANGLES)
/ENTRY is not supported with /QUERY=(PRODUCT,IMAGE)

/OUTPUT

The /OUTPUT qualifier allows the user to redirect the output to a file on disk. Users must also indicate a specific dictionary. /OUTPUT may be used with all dictionaries.

Example:

```
$SDB/OUT=myfile.txt/QUERY=IMAGE
```

will display all entries in the image dictionary. The output is redirected to a file in the working directory called myfile.txt.

UNSUPPORTED COMBINATIONS:

/FULL is not supported

/BEFORE

The /BEFORE and /AFTER qualifiers allow the user to specify a range of entry numbers. Users must also indicate a specific dictionary. /BEFORE and /AFTER may be used with all dictionaries.

Example:

```
$SDB/AFTER=100/BEFORE=900/QUERY=(CORE,IMAGE)
```

will display all entries in the image and core dictionaries between entry numbers 100 and 900, inclusive.

UNSUPPORTED COMBINATIONS:

/BEFORE and /AFTER is not supported with /LOCATION

/AFTER

The /BEFORE and /AFTER qualifiers allow the user to specify a range of entry numbers. Users must also indicate a specific dictionary. /BEFORE and /AFTER may be used with all dictionaries.

Example:

$SDB/AFTER=100/BEFORE=900/QUERY=(CORE,IMAGE)

will display all entries in the image and core dictionaries between entry numbers 100 and 900, inclusive.

UNSUPPORTED COMBINATIONS:

/BEFORE and /AFTER are not supported with /LOCATION

/DISKS

The /DISKS qualifier creates a listing of all fixed and removable disk storage containing registered data. /DISKS may only be used in the format below.

$SDB/DISKS

will display all disks entered in the database.

UNSUPPORTED COMBINATIONS:

ALL OTHER COMBINATIONS ARE UNSUPPORTED.

/DAYS

The /DAYS qualifier creates a listing of all dates which are entered into the database. /DAYS may only be used in the format below.

$SDB/DISKS

will display all dates entered in the database.

UNSUPPORTED COMBINATIONS:

ALL OTHER COMBINATIONS ARE UNSUPPORTED.

Appendix F: Example Input Parameters File For the GMS Satellite

In addition to the common parameters defined below, the input parameter file may contain other parameters that are satellite platform and/or computer platform specific.

Parameter	Meaning
GMS_DIR=/users/ingest/data/gms/	Pathname of GMS ingest area
SDISK=/users	Source disk location
SDIR=/ingest/data/gms/	Source directory location
TDIR=/sercaa/data/	Target directory location
QUE=GIANT_BATCH	Batch queue used for registration
SDB_SET=NOV93	SERCAA 10-day save set id
START_DAY=93325	Starting year and julian day of data to ingest
TDISKA=arcopt$dka100:	Target disk for registered data
TLABELA=GMS001	Label of volume mounted on target disk.
TLOGA=GMS$001	Logical name to refer to target disk
TDISKB=arcopt$dka200:	Second target disk for registered data (if needed)
TLABELB=GMS003	Same as above
TLOGB=GMS$003	Same as above
TDISKC=arcopt$dka300:	Third target disk for registered data (if needed)
TLABELC=GMS005	Same as above
TLOGC=GMS$005	Same as above

Appendix G: Example SDB Information File for the GMS Platform

This file is responsible for directing the registration of data into the SERCAA database. The entries section (the last section in the file) is appended to the file once the entry numbers are determined. After the entry numbers are recorded, the SDB information file serves as an audit trail should an operator be required to manually re-register the data due to a failure.

[SOURCE]

The Source section specifies the location of source files.

SDISK:=user_aer:
SDIR:=[ingest.data.gms]

[TARGET]

The Target section specifies the location of target files and the appropriate SDB database set to use.

TDISK:=arcopt$dka300:
TDIR:=[sercaa.data.993327]
TLABEL:=GMS002
TLOG:=GMS$002
START_DAY:=93325
SDB_SET:=NOV93

[CORE]

The Core section contains parameters necessary to fill core data dictionary entries.

SAT_CODE:=22
NUM_CHAN:=2
SENSOR_CODE_1:=30
SENSOR_CODE_2:=31
AREA:=set

[SATIMG]

The Satimg section contains parameters necessary to fill image data dictionary entries.

ALPHA_CYYJJJ:=993327
ALPHA_HH:=19
ZULU_YYJJJ:=93327
ZULU_HHMMSS:=193700
NUM_LINES:=794
ELEM_PER_LINE:=926
BYTES_PER_ELEM:=1
IMAGE_TYPE:=1
RESOLUTION:=5.000000
FILE_IMG_1:=g04_001_set_327_19.tif
FILE_IMG_2:=g04_002_set_327_19.tif

[LATLON]

The Latlon section contains parameters necessary to fill latlon data dictionary entries.

LL_REC_LEN:=96
LL_LINE_INTERVAL:=1
LL_ELEM_INTERVAL:=40
LL_ELEM_PER_LINE:=24
LL_SHIFT:=1
NUM_LL:=1
FILE_LL_1:=g04_lat_set_327_19.dat

[ANGLES]

The Angles section contains parameters necessary to fill angles data dictionary entries.

ANG_REC_LEN:=288
ANG_LINE_INTERVAL:=1
ANG_ELEM_INTERVAL:=40
ANG_ELEM_PER_LINE:=24
ANG_VIEWING:=1
ANG_ZENITH:=1
ANG_AZIMUTH:=1
ANG_SHIFT:=1
NUM_ANG:=1
FILE_ANG_1:=g04_ang_set_327_19.dat

[ENTRY]

The entries section records the entry numbers that were assigned to the above entries.

ENTRY_1:= 5310
ENTRY_2:= 5311
ENTRY_3:= 5312
ENTRY_4:= 5313

SDB_Get_Data

SDB_Get_Data is an unsigned integer function used to retrieve satellite image data from the SERCAA database.

Format SDB_Get_Data entry, min_sample, max_sample,min_line,max_line, select,array

Returns Status return values are listed under Condition Values Returned.

Arguments:

entry

usage: word_signed
type: word integer (signed)
access: read only
mechanism: by reference

entry is the database entry number associated with the data to be retrieved.

min_sample

usage: word_signed
type: word integer (signed)
access: read/write
mechanism: by reference

min_sample is the minimum x area coordinate (one relative) within the image to begin the retrieval. An input value of zero instructs the routine to use the coordinate from the database.

max_sample

usage: word_signed
type: word integer (signed)
access: read/write
mechanism: by reference

max_sample is the maximum x area coordinate (one relative) within the image to end the retrieval. An input value of zero instructs the routine to use the coordinate from the database.

min_line

usage: word_signed
type: word integer (signed)
access: read/write
mechanism: by reference

min_line is the minimum y area coordinate (one relative) within the image to begin the retrieval. An input value of zero instructs the routine to use the coordinate from the database.

max_line

usage: word_signed
type: word integer (signed)
access: read/write
mechanism: by reference

max_line is the maximum y area coordinate (one relative) within the image to begin the retrieval. An input value of zero instructs the routine to use the coordinate from the database.

select

Usage: char_string
type: character string
access: read only
mechanism: by reference

select specifies the month/year of the SERCAA 10-day save set of interest. **select** contains the address of a character string referencing this string. The string format is MMMYY.

array

usage: structure
type: structure
access: write only
mechanism: by reference

array is a structure containing a reference to the returned data. The structure is composed of the following elements:

> **data** is an unsigned character pointer to the data. If the return status from SDB_Get_Data is fatal, this field is undefined.

Description:

SDB_Get_Data is an integer function that retrieves satellite image data from the SERCAA database. Input parameters specify starting offsets and the number of lines and samples to take starting at the offsets. Zero offsets imply the user is requesting the entire image. If the specified image subsection partially exceeds the image bounds, the user's buffer will be zero-filled as needed on a line-by-line basis. The offsets and dimensions

upon return will reflect the area within the user buffer that contains valid data. The function variable will also contain a status value indicating a change in dimensions of valid data. The buffer area for data is allocated by the routine and passed back to the user in the return structure **array**. SDB_Get_Data recognizes and handles raw and TIFF file formats.

Restrictions apply to requests that define a subset (or block) of the image. First, the initial call to SDB_Get_Data regulates the block size, future calls cannot deviate from the initial block size. Secondly, block access is sequential. Direct access to records within the file is not supported.

SDB_Get_Data uses a linked list to manage requests. The list uses entry number and the buffer area to arbitrate separate requests. This allows the user to retrieve data from multiple image files within the same program or to make multiple requests to the same image file with different block sizes.

SDB_Get_Data makes extensive use of dynamic memory. It is important that this memory be freed once the data returned by SDB_Get_Data is no longer needed so that the calling process can reuse the virtual memory. This is done by calling the function **SDB_Free_Data** (refer to attached documentation). In addition to freeing memory, this routine also closes the image file and removes the associated entry from the linked list.

<u>Example Code Stub in C:</u>

```
#include "sdb_access.h"
#include "sdb_errors.h"
#include "vms_mount.h"

main()
{

        short           entry;
        short           min_sample;
        short           max_sample;
        short           min_line;
        short           max_line;

        ARRAY           array;

        unsigned int    status;

        char            select[] = "NOV93";
        char            device[] = "ARCOPT$DKA100:";
        char            label[] = "GMS002";
        char            logical[] = "GMS$002";

        int             i;
        int             total_data;

/* Initialize array structure */

        status = sdb_init_array(&array);

/* Fill in input arguments */

        entry = 900;
        min_sample = 0;
        max_sample = 0;
        min_line = 0;
        max_line = 0;

/* Mount the device that holds the data */

        status = vms_mount(&device[0], &label[0], &logical[0]);
        if (status == FAIL) {
         printf ("Failure on call to VMS_Mount\n");
         exit(1)
        }

/* Get the data */

        status = sdb_get_data(&entry,
                              &min_sample,
                              &max_sample,
                              &min_line,
```

```
                                &max_line,
                                &select[0],
                                &array);
        if (status != PASS) {
         printf("Failure on call to SDB_Get_Data\n");
         exit(1);
        }

        total_data = max_sample * max_line;

        for (i=0;  i<total_data;  i++) {
         printf("%d\n",array.data[i]);
        }
/*  Free resources  */

        status = sdb_free_data(&entry, &array);
        if (status == FAIL) {
         printf("Failure on call to SDB_Free_Data\n");
        }

/* Dismount the device  */

        status = vms_dismount(&device[0]);
        if (status == FAIL) {
         printf ("Failure on call to VMS_DisMount\n");
         exit(1)
        }

}
```

Condition Values Returned:

PASS -- Normal completion status value.

FAIL -- An error occurred in client/server communications.

SDB_CHGDIM -- Sample and/or line dimensions have changed. Check the return values.

SDB_ITEMNOTFOUND -- An item could not be found in the linked list for this call.

SDB_ALLOCFAIL -- An attempt to dynamically allocate virtual memory failed.

SDB_NOTINDATA -- The data request was out of range.

SDB_BADFT -- The file type is invalid.

SDB_FOERR -- The file could not be opened.

SDB_BADSTRIP -- Strip was out of sequence.

SDB_MINGTMAX -- Minimum sample/line was greater than maximum sample/line.

SDB_READERR -- A "C" I/O file read error was encountered.

Appendix I: SERCAA Database Query Routine (RPC version)

SDB_Query_Core

SDB_Query_Core is an unsigned integer function used to query the SERCAA core data dictionaries for entries that satisfy input query criteria.

Format SDB_Query_Core function, query, address, number_entries, select

---

Returns Status return values are listed under Condition Values Returned.

Arguments:

function

usage: longword_signed
type: longword integer (signed)
access: read only
mechanism: by value

function controls the function performed by SDB_Query_Core. The only function currently supported is to perform a query on the core data dictionary. The **function** argument is a longword value containing a pre-defined symbolic constant. The constant is:

SDB__Query_Core

query

usage: structure
type: structure
access: read only
mechanism: by reference

query is a structure containing the query parameters used to perform the search for core entries that satisfy the query criteria. The elements of this structure are:

start_yyjjjhhmm is a signed longword containing the start date-time of the query in YYJJJHHMM format or 0 (NULL) to indicate a wild card operation.

stop_yyjjjhhmm is a signed longword containing the stop date-time of the query in YYJJJHHMM format or 0 to indicate a wild card operation. If not 0, must be equal to greater than **start_yyjjjhhmm.**

satellite is a signed longword containing the satellite code as defined in the SERCAA database or 0 for a wild card operation.

location is a signed longword containing the location code or 0 for a wild card operation. **location** is specified by the use of symbolic constants. They are:

SDB__SAT
SDB__SDT
SDB__SET

match_flag is a signed longword containing the match criteria to use for the starting date-time. **match_flag** is specified by the use of symbolic constants. They are:

EXACT	attempt an exact match on the start date-time specified in **start_yyjjjhhmm.** ***(Default)***
CLOSEST	find the closest match to **start_yyjjjhhmm.**
CLOSEST_NOOVER	find the closest match to **start_yyjjjhhmm** without exceeding it.

address

usage: structure
type: structure
access: write only
mechanism: by reference

address is a structure containing information on the core entry structures that satisfied the search criteria. The structure contains the following element:

> **core** is an indirect reference (double pointer) to an array of pointers. The pointers reference the core entry structures that satisfied the search criteria. If **number_entries** is zero this field is undefined.

number_entries

usage: longword_unsigned
type: longword integer (unsigned)
access: write only
mechanism: by reference

number_entries is the total number of entries found that satisfied the query criteria. **number_entries** also indicates the size of the return pointer list. **number_entries** can range from 0 (no entries satisfied the query criteria) to the total number of entries in the core data dictionary.

select

Usage: char_string
type: character string
access: read only
mechanism: by reference

select specifies the month/year of the SERCAA 10-day save set of interest. **select** contains the address of a character string referencing this string. The string format is MMMYY.

Description:

SDB_Query_Core is a programming interface to sorted core data dictionaries maintained in parallel with the keyed core data dictionaries used for SERCAA. The sorted core data dictionaries are triply sorted on the parameters defined in the QUERY structure used in this function, namely: date-time, satellite id and location id. This function is designed to retrieve entries that satisfy the query criteria setup by the programmer in the QUERY structure. Note that any combination of specified and non-specified (i.e. wild card operation) query parameters is possible. If all parameters are wild carded, all entry numbers will be returned. Unlike its VAX-VMS counterpart, the RPC version of SDB_QUERY_CORE makes the return of memory references to the actual entry structures a default action rather than an option.

Example Code Stub in C:

```
#include "sdb_query_core.h"
#include "sdb_errors.h"
#include "sdba.h"

main()
{
        QUERY          query;
        ADDR           addr;

        int            fun;
        int            i;
        unsigned int   num_entries;
        unsigned int   status;

        char           select[] = "MAY93";

/* Initialize address structure */

        status = sdb_init_addr(&addr);

/* Fill in query structure. Request is all NOAA-11 data for the case day 93145. */

        fun = SDB__Query_Core;
        query.start_yyjjjhhmm = 931450000;
        query.stop_yyjjjhhmm = 931452359;
        query.satellite = NOAA_11;
        query.location = 0;
        query.match_flag = CLOSEST;

/* Get entry numbers that satisfy query criteria parameters */

        status = sdb_query_core(fun, &query, &addr, &num_entries, &select);

        if (status != PASS) {
         printf("Failure on call to SDB_Query_Core\n");
         exit(1);
        }

/* Loop thru number of found entries. Each time thru the loop dereference a pointer to
   an entry structure and access the contents of the structure */

        for (i=0; i<num_entries; i++) {
         printf("\n Core entry is %d \n", addr.core[i]->entry);
         printf("\n Core data type is %d \n", addr.core[i]->data_type);
         printf("\n Core satellite code is %d \n", addr.core[i]->sat_code);
         printf("\n Core sensor code is %d \n", addr.core[i]->sensor_code);
         printf("\n Core country code is %s \n", addr.core[i]->country_code);
         printf("\n Core data file is %s \n", addr.core[i]->data_file);
        }
}
```

Condition Values Returned:

PASS -- Normal completion status value.

FAIL -- A failure occurred in client/server communications.

SDB$FOERR -- A file open error occurred on the sorted core data dictionary file.

SDB$READERR -- A read error occurred on the sorted core data dictionary file.

SDB$ALLOCFAIL -- An attempt to dynamically allocate virtual memory failed.

SDB$COREACCESSFAIL -- An error occurred while processing data from the keyed core data dictionary file.

SDB$DTCALCFAIL -- An error occurred while making a date-time calculation. This is usually the result of a corrupt date-time within a data dictionary.

SDB$DTOUTOFRNG -- The start and/or stop date-time parameter(s) in the QUERY structure are/is out of the range of the data dictionary.

VMS_Mount

VMS_Mount is an unsigned integer function used to mount a VMS device on behalf of the calling client.

Format VMS_Mount device, label, logical

Returns Status return values are listed under Condition Values Returned.

Arguments:

device

usage: char_string
type: character string
access: read only
mechanism: by reference

device is the device name of the device to mount. **device** can range from 1 to 64 characters (case insensitive) and can reference a physical name or a logical name. If a logical name is used it must translate to a physical device name.

label

usage: char_string
type: character string
access: read only
mechanism: by reference

label is the label given to the volume when it was initialized. **label** references a one to twelve character string (case insensitive).

logical

usage: char_string
type: character string
access: read only
mechanism: by reference

logical specifies a logical name for the volume. **logical** can range from 1 to 64 characters (case insensitive).

Description:

VMS_Mount is used to mount a VMS device on behalf of the calling client. VMS_Mount can mount mass storage devices that support the VMS Files-11 on-disk structure. This includes devices such as hard disk, magneto-optical drives and CD-ROM drives. VMS_Mount mounts the device shared so other users/applications can access the volume simultaneously. When finished with the device, VMS_DisMount must be called to properly manage mount counts on the device (see attached documentation on VMS_DisMount).

<u>Example Code Stub in C:</u>

See example in SDB_Get_Data

Condition Values Returned:

PASS -- Normal completion status value.

FAIL -- A failure occurred in client/server communications or a VMS error occurred.

VMS_DisMount

VMS_Mount is an unsigned integer function used to dismount a VMS device on behalf of the calling client.

Format VMS_DisMount device

Returns Status return values are listed under Condition Values Returned.

Arguments:

device

usage: char_string
type: character string
access: read only
mechanism: by reference

device is the device name of the device to dismount. **device** can range from 1 to 64 characters (case insensitive) and can reference a physical name or a logical name. If a logical name is used it must translate to a physical device name.

Description:

VMS_DisMount dismounts a device previously mounted with VMS_Mount. If the calling client was the only application to have mounted the device, the device will be physically dismounted. Otherwise, the mount count is simply decremented.

Example Code Stub in C:

See example in SDB_Get_Data

Condition Values Returned:

PASS -- Normal completion status value.

FAIL -- A failure occurred in client/server communications or a VMS error occurred.